

AD-A163 271 STARS (SOFTWARE TECHNOLOGY FOR ADAPTABLE AND RELIABLE 1/1

AD-A163 271 STARS (SOFTWARE TECHNOLOGY FOR ADAPTABLE AND RELIABLE 1/1

AD-A163 271 STARS (SOFTWARE TECHNOLOGY FOR ADAPTABLE AND RELIABLE SYSTEMS) METHODOLOG. (U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C W McDONALD ET AL. MAR 85

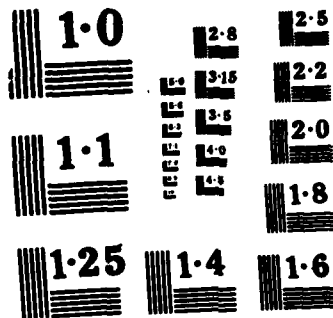
ALEXANDRIA VA C H MCDONALD ET AL. MAR 85

UNCLASSIFIED IDA-P-1814-VOL-2 IDA/HQ-85-29622

UNCLASSIFIED IDA-P-1814-VOL-2 IDA/HQ-85-29622 F/G 9/2

UNCLASSIFIED IDA-P-1814-VOL-2 IDA/HQ-85-29622 F/G 9/2 NL

[illegible][illegible]



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

(2)

IDA PAPER P-1814

AD-A163 271

STARS METHODOLOGY AREA SUMMARY

Volume II: Preliminary Views on the Software Life Cycle
and Methodology Selection

Catherine W. McDonald
William Riddle
Christine Youngblut

DTIC
ELECTE
JAN 22 1986
S D

March 1985

Prepared for

Office of the Under Secretary of Defense for Research and Engineering

DTIC FILE COPY



INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, VA 22311

86 1 22 061

IDA Log No. HQ 85-29622

The work reported in this document was conducted under Contract No. MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) P-1814 Volume II			7a. NAME OF MONITORING ORGANIZATION DoD-IDA Management Office		
6a. NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses		6b. OFFICE SYMBOL (If applicable)	7b. ADDRESS (City, State, and ZIP Code) 1801 N. Beauregard Street Alexandria, VA 22311		
6c. ADDRESS (City, State, and ZIP Code) 1801 N. Beauregard Street Alexandria, VA 22311		7a. NAME OF MONITORING ORGANIZATION DoD-IDA Management Office			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION OUSDRE (R&AT) STARS JPO		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031		
8c. ADDRESS (City, State, and ZIP Code) 1211 Fern Street Arlington, VA 22202		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
				T-5-293	
11. TITLE (Include Security Classification) STARS Methodology Area Summary Volume II: Preliminary Views on the Software Life Cycle and/ Methodology Selection					
12. PERSONAL AUTHOR(S) Catherine W. McDonald, William Riddle, Christine Youngblut					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Year, Month, Day) March 1985	
				15. PAGE COUNT 79	
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	methodology, Ada, software engineering, methods, life cycle, classification, evaluations, selection, software, characteristics, maintenance design		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Initially, the Ada Joint Program Office (AJPO) sponsored Professors Peter Freeman and Anthony Wasserman to identify requirements for software development methodologies that would allow the Department of Defense (DoD) to realize the full potential of Ada. Since that time, the work on methodologies to support Ada has been transferred to the DoD Joint Program Office for the program entitled Software Technology for Adaptable and Reliable Systems (STARS). The STARS Joint Program Office (SJPO) objective is to improve the productivity level of software system development and support as well as the resulting quality of deployed software systems. This report consists of two volumes: Volume I presents the organization and plans of the STARS Methodology Coordination Team. Volume II is a technical report concerned with the development of methodology classification, evaluation and selection technologies and a framework of characteristics that can be used to support these technologies.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION		
22a. NAME OF RESPONSIBLE INDIVIDUAL			22b. TELEPHONE (Include Area Code)		22c. OFFICE SYMBOL

IDA PAPER P-1814

STARS METHODOLOGY AREA SUMMARY

Volume II: Preliminary Views on the Software Life Cycle and Methodology Selection

Catherine W. McDonald
William Riddle
Christine Youngblut

March 1985

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	



INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-4-222



ACKNOWLEDGEMENTS

The authors would like to thank those people who participated in the limited review of this document. Suggestions not incorporated in this document will be addressed in future publications under the STARS Methodology Area.

The authors also appreciate the efforts of Joe Batz, Carol Morgan and Robert Mathis from the STARS Joint Program Office, and Tom Probert and Jack Kramer of IDA for their comments on previous versions of this document. Special thanks go to Lou Chmura, Sam Redwine, Pete Fonash, George Sumrall and all the other members of the MCT who contributed so heavily to the material presented here.

Finally, the authors would like to thank Jo Ann Stilley and Betty Henderson for their assistance in preparing the document. Mrs. Stilley spent many hours typing the various drafts and the final version. Ms. Henderson prepared all of the figures and tables.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	111
LIST OF FIGURES	vii
1.0 INTRODUCTION	1-1
2.0 SOFTWARE METHODOLOGY CONCEPTS	2-1
2.1 Basic Concepts	2-2
2.2 Software Versions and Levels of Concern	2-3
2.3 Software Life Cycles	2-5
2.4 Software Methodology	2-9
2.5 Classes of Methodologies	2-10
2.5.1 Product-Oriented Methodologies	2-11
2.5.2 Other Methodology Classes	2-13
2.5.3 Mixtures of Methodologies	2-14
2.6 Summary	2-14
3.0 CLASSIFICATION, EVALUATION AND SELECTION	3-1
3.1 Overview	3-1
3.2 Focusing on Ada-Compatible Methodologies	3-3
3.3 Selection Technology	3-3
3.4 Evaluation Technology	3-6
3.5 Classification Technology	3-8
3.6 Development of the Technology	3-10
4.0 METHODOLOGY CHARACTERISTICS FRAMEWORK	4-1
4.1 Overall Structure of the Characteristics Framework	4-1
4.2 Lower Level Structure of the Characteristics Framework	4-3
4.3 Populating the Framework with Characteristics	4-5
4.4 Defining the Scope of the Other Categories	4-11
4.5 Current Status of the Characteristics Framework	4-16
5.0 SUMMARY	5-1
REFERENCES	R-1
APPENDIX A GLOSSARY	A-1

LIST OF FIGURES

	Page
Figure 2-1 Activities During a Version Life Cycle	2-6
Figure 2-2 Activities During a Software System Life Cycle	2-8
Figure 2-3 Relationship Between Software System Life Cycle Phases and Life Cycle Activities	2-12
Figure 3-1 Selection, Evaluation and Classification Technologies	3-2
Figure 3-2 Selection, Evaluation and Classification for Ada-compatible Methodologies	3-4
Figure 3-3 Selection Technology	3-5
Figure 3-4 Evaluation Technology	3-7
Figure 3-5 Classification Technology	3-9
Figure 3-6 Classification, Evaluation and Selection Technologies for Ada-compatible Methodologies Including Secondary Dependencies	3-12
Figure 4-1 Definition of Characteristics Categories	4-2
Figure 4-2 Characteristics Framework	4-4
Figure 4-3 Preliminary Matrix for Collections of Technical Characteristics	4-6
Figure 4-4 Technical Characteristics Pertaining to Efficiency and Product-oriented Methodologies	4-8
Figure 4-5 Preliminary Matrix for Collections of Management-Related Characteristics	4-13

LIST OF FIGURES (Continued)

		Page
Figure 4-6	Preliminary Matrix for Collections of Usage-Related Characteristics	4-14
Figure 4-7	Preliminary Matrix for Collections of Ada-compatibility Characteristics	4-15

1.0 INTRODUCTION

In September 1983, the Methodology Coordination Team (MCT) began addressing the comments made on the METHODMAN document prepared by Peter Freeman and Anthony Wasserman (1). These comments included:

- only "traditional" methodologies were addressed
- it focused primarily on the development part of the software life cycle,
- no relationship was established to the emerging DoD software life cycle defined in DoD-STD-SDS,
- the set of characteristics given for classifying software methodologies was incomplete and many of the identified characteristics were not concrete enough to be measured,
- the organization of the set of characteristics was ad hoc, and
- the requirements given for Ada*-compatible methodologies were too general and not specifically related to the characteristics.

In addressing these comments, the MCT focused its attention on two major issues: (1) software development and 'maintenance' life cycle and (2) an approach to classifying, evaluating and helping people to select methodologies. This report is a summary of the initial work of the MCT on these issues.

Basic concepts of software creation and evolution are identified in Section 2. The use of modeling, information accumulation and analysis are discussed as ways of coping with the risk, uncertainty and complexity associated with most software projects. Different types of software versions are distinguished and the general nature of the methodological issues associated with each are discussed. The concept of a life cycle is defined and then used to identify the spectrum of activities involved in creating and evolving various types of software versions. The concept of using specific methods or general methodologies to organize and discipline these activities is then discussed and several different types of methodologies are distinguished.

* Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office)

Finally, the prospects for covering a broad spectrum of activities by using several compatible methodologies are discussed.**

In Section 3, the problem of evaluating and selecting among alternative methodologies is addressed. The need for technologies to support the classification and evaluation of specific methodologies and the selection of a methodology from among alternatives is established. Interrelationships among these technologies are addressed, as is the process of co-evolving them. The intertwined roles of selection criteria, evaluation measures and classification metrics are discussed. Finally, the discussion establishes the critical need for detailed methodology characteristics supporting the definition of criteria, measures and metrics and therefore the classification, evaluation and selection technologies.

A means of organizing the potentially very large set of detailed characteristics is the subject of Section 4. A preliminary framework is proposed for organizing the characteristics such that one can identify a subset of characteristics pertinent to a general area of methodological concern, a specific type of software version or a general methodology. The framework specifies a way of further organizing any subset of characteristics, in particular, subsets identified by using the gross structural framework. The framework is specifically designed to be extensible and this aspect is discussed. In addition, a procedure is presented for enumerating characteristics to provide an initial population which could then be refined and elaborated through further use of the enumeration procedure or other procedures.

The scope of the work presented here is broad and its nature is preliminary. It is intended to give direct and balanced attention to both the software creation and evolution process and the products produced during this process. It is also intended to encompass not only software programs themselves but also all the myriad other documents and products produced during a software project as well as the activities concerning the role of software as part of some automated system. While the work has been, of necessity, developed in the context of extant methodologies, an

** Throughout this report, and particularly in Section 2, care is taken to establish a well-defined terminology. A glossary appears as Appendix A. In preparing definitions the IEEE Glossary of Software Engineering (2), and other glossaries, were consulted. Our definitions deviate from these already established definitions whenever it was felt necessary.

attempt has been made to provide a groundwork which is flexible enough to accommodate future improvements in software methodology as they appear. Overall, the work is intended to be a preliminary, malleable step toward obtaining and using the specific algorithms, metrics, attributes and data that can be expected in the future.

2.0 SOFTWARE METHODOLOGY CONCEPTS

Discussions of software methodology are often plagued by confusion and misunderstanding because of differing perceptions and experiences. For example, software developers and project managers usually have quite different views of the software creation and evolution process because of the different activities they see as making up the process and the quite different concerns they must address. The situation is often complicated, as with many aspects of computer science, by a lack of a widely-used, consistent terminology.

Terminology and a way of thinking about the software creation and evolution process are presented in this section to provide a conceptual basis for subsequent discussion of methodological issues. The way of thinking attempts to focus on the process itself as well as on the products produced as a result of the process, to be pertinent to new approaches as well as those currently in vogue, and to tie the issues of software creation and evolution into the larger problem of creating and evolving the automated system of which the software is a part.

The following statements provide a quick synopsis of the conceptual basis and terminology introduced in this section:

1. A software method is a disciplined process for producing software. It assists in coping with the high levels of uncertainty, complexity and risk that surround most software projects. The majority of current software methods rely on modeling, information accumulation and analysis techniques to provide this assistance.
2. A software methodology is a collection of methods. The collection may serve to highlight those aspects common to all the methods. Or it may serve to define an approach to software development and post-deployment support which is broader in life cycle coverage than any of the methods. In either case, a specification for the methodology defines those general principles, practices and procedures shared by all of the methods rather than the specific details peculiar to any particular method.

3. There are several levels of concern when considering software methodologies. These range from the software being one part of an automated system to the software being an object undergoing change to correct deficiencies or enhance its capabilities. Methodological concerns vary across these levels.
4. The software life cycle organizes the activities performed during software development and post-deployment support. The phases of a life cycle differ, in terms of the emphasis on the activities involved, across the levels of concern.
5. Product-oriented methodologies organize development and post-deployment support activities, each emphasizing the production of (intermediate or final) products needed to achieve some milestone. Other types of methodologies reflect other approaches such as developing a sequence of gradually more mature versions.

2.1 Basic Concepts

Software creation and evolution involves the preparation of various products by following some process. One product is the software's code, that is, its executable version. Other, equally important, products are designs and specifications for the software, users' manuals, test case definitions and results, and project histories (currently prepared in document form). In this report, the term software refers to all such products, not just the executable code.

Most DoD software systems have a level of complexity that is almost overwhelming. In addition, creating or evolving the software system is often a high-risk activity because the software is frequently targeted for use in an application where little experience exists to indicate whether an acceptable result can be obtained in a meaningful time period and with a reasonable expenditure. This risk is accompanied by a high level of uncertainty about the system's functionality and performance, an uncertainty that persists at least until the software is implemented and can be tested in its operational environment. There are many techniques for coping with this complexity, uncertainty and risk. Current approaches to software creation and evolution tend to emphasize three techniques: modeling, information accumulation and analysis.

Every product of the software development process, except the software's executable load module, is a model, that is, an abstract description not containing all of the details. The obvious benefit of a model is that it helps in coping with complexity by highlighting pertinent aspects of the software. Models also aid in

coping with risk since they provide points which can be returned to should errors occur when

determining the details. Risk can be addressed by gradually and rationally elaborating the details of a software system in a way that supports periodic assessment of progress, that is, by following a process of information accumulation. This process can vary from being informal and intuition-based to being rigorously defined in terms of specific activities, techniques, work products and reviews. Finally, analysis is the process of assessing the software's suitability and, as such, primarily addresses the uncertainty problem. Techniques for incrementally testing portions of a software system as they are constructed are prime examples.

Modeling, information accumulation and analysis provide a strategic basis for coping with complexity, uncertainty and risk. These and other techniques require the use of software technology and software methodology. Software technology provides basic techniques for accumulating information as a series of well-defined models that can be analyzed before the software is fully operational. Software methodology imposes the discipline making the overall process orderly and ensuring smooth and steady progress. It also helps assure that project resources are appropriately used so as to arrive at closure on time and within budget.

Software technology and software methodology are highly interrelated, with the details of each having a strong impact on the other. Consequently, while focusing on software methodology, it is important to realize that fully working out the details of a particular approach to disciplining the process requires identification of the software technology which supports, and complements, the approach.

2.2 Software Versions and Levels of Concern

Software must be frequently changed to accommodate changes in requirements, repair errors, or upgrade quality. This leads to several versions of the software existing over time. Some of these versions are transient attempts to meet the requirements while others have relatively long lifetimes of operational service. Versions actually placed into service may reflect relatively minor modification or they may reflect the major modifications needed to provide similar functionality in totally different operational situations.

Three major categories of versions are of interest because they relate to different sorts of methodological concerns. A software system is a version which delivers a capability in a form appropriate for integration with other components to create an automated system. For example, different software systems may

deliver a flight control capability for different types of aircraft. A software release is a version of a software system that is incorporated into and supported as part of an operational automated system. Each release is intended to meet the software system's requirements and new releases appear primarily because of changes to the required functionality and performance or corrections to remove errors discovered during operational use. A software variant is a (perhaps incomplete) version of a software release that is an (perhaps incorrect) attempt to meet a release's set of requirements. Successive variants may reflect design and implementation changes made to bring the software into closer conformance to the release's set of requirements.

These three major categories are hierarchically related. A release encompasses a sequence of variants, each being a more suitable attempt to meet the requirements established for the release. Similarly, a software system encompasses a sequence of releases meeting the changing requirements levied by the software system's operational environment.

This distinction of three levels of software versions delineates three levels of concern for organizing and disciplining the process of software creation and evolution.

Corresponding to the highest level (software systems) are the overall concerns of how software fits into the automated system of which it is a part. The process of software creation and evolution at this level must account for activities such as: pre-software definition and design of the automated system itself, making tradeoff decisions concerning which components will be realized in software, integration of the separately created and evolved components, upgrading the requirements levied against a software system, and demonstrating the software system's validity in terms of meeting recognized needs.

Corresponding to the intermediate level (software releases) are the project management concerns of delivering a capability on time and within budget. The process at this level must account for activities such as: deciding when a new release is warranted and can be undertaken, delivery of the individual releases on time and within budget, managing the overall process using tools for change control, traceability, impact analysis, etc., and modularization of the software system into major subsystems.

And, finally, corresponding to the lower level (software variants) are the technical concerns relating to preparing a complete and demonstrably suitable version which meets a set of requirements. The process at this level must account for activities such as: periodically verifying that the requirements are met, and determining low-level modules that can be worked on by individuals or small groups and that lead to an efficient implementation.

2.3 Software Life Cycles

Every software version will be created and evolved through activities which make up the version's life cycle. The life cycle starts once the need for the version is recognized and continues until the version is retired from service.

Many models of the life cycle have evolved over time. The majority of these primarily pertain to software releases and attempt to discipline the life cycle. The majority, therefore, are strongly related to only a portion of the methodological issues (those corresponding to releases) and are prescriptive, or at least reflective, of a particular approach to software creation and evolution. A more generic view of the life cycle, pertaining to all types of versions and relatively independent of any particular software creation and evolution approach, is presented in this section and used to define various terms describing activities during software creation and evolution.

Regardless of how the creation and evolution process is carried out, every version will pass through a sequence of historical time points during its life cycle. The time points in the history of many types of versions are shown in Figure 2-1 and can be defined as follows:

- Conception: the first point at which a need for a version is recognized,
- Definition: presentation of a possibly rough and incomplete statement of the problem to be solved by the version
- Specification: presentation of a possibly rough and incomplete description of the user-visible features of the version,
- Delivery: presentation of a believed-to-be-suitable version for integration into the automated system
- Deployment: presentation of the version for actual use,
- Freezing: determination that no further changes will be made to the version, and
- Retirement: removal of the version from service.

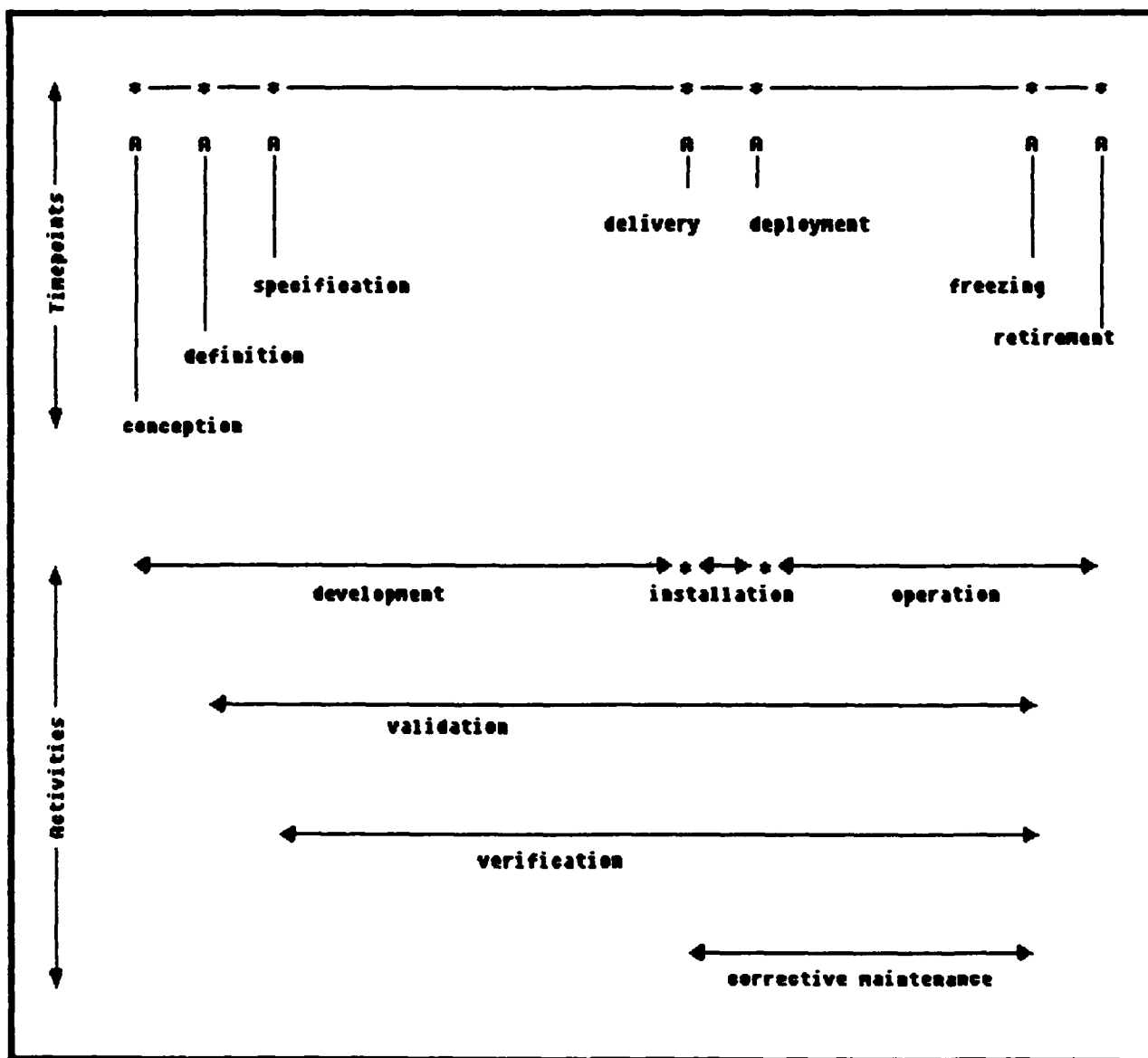


Figure 2.1: Activities During a Version Life Cycle

These time points can be used to define the following activities:

Development:	the activity of preparing a deliverable version
Installation:	the activity of preparing a deployable version from a delivered version,
Operation:	the activity of using a deployed version,
Validation:	the activity of analyzing a version to assure that it meets user needs,
Verification:	the activity of analyzing a version to assure that it meets its requirements,
Corrective Maintenance:	the activity of evolving the version, after deployment, to correct deficiencies.

This view of the life cycle is pertinent, without modification, to software releases since it has a strong heritage in existing life cycle models. It is also pertinent to software variants once one realizes that a variant's life cycle is a subset of this general life cycle because a variant may inherit much of its detail (in particular, its specification) from a previous variant and may be dropped from active consideration prior to delivery.

The life cycle of Figure 2.1 is pertinent to a software system although some additional terms are traditionally used when describing a software system's life cycle. These terms are introduced in Figure 2.2. The time lines at the top of this figure represent the life cycles of the various releases occurring during the software system's life cycle. The release life cycles may overlap, except that: 1) the delivery and deployment time points for the successive releases are ordered over time, and 2) the deployment of one release is generally coincident with, or prior to, the retirement of the previous release.

The new terms introduced in Figure 2.2 are:

Operation & Maintenance:	the activity subsequent to deployment of the initial release of a software system, and
Perfective and Adaptive Maintenance:	the activity of upgrading a software system through new releases.

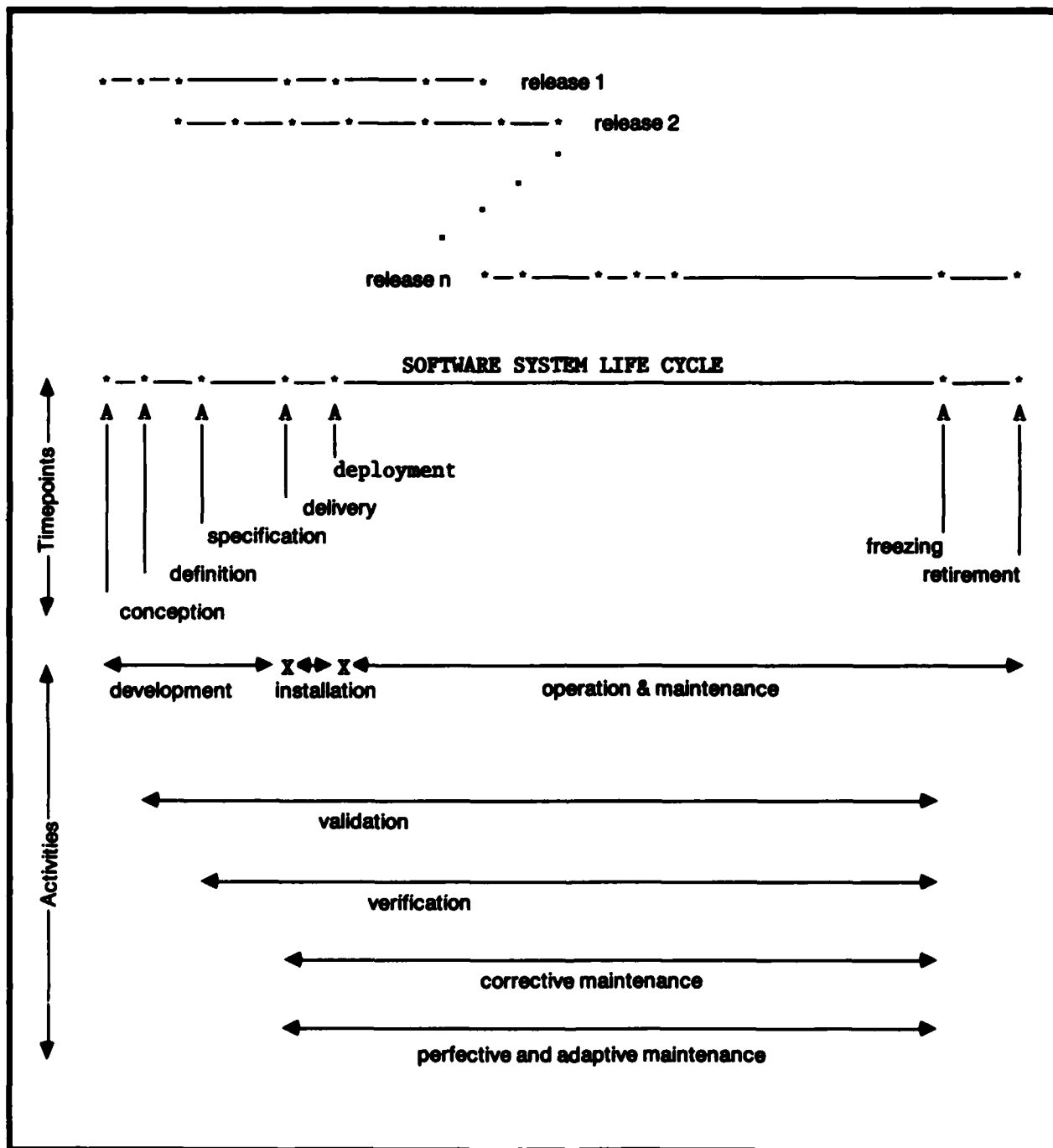


Figure 2.2: Activities During a Software System Life Cycle

The first is just a more explicit name for "software system operation" and reflects common terminology. The second reflects the additional activity of preparing new releases to enhance a software system's capabilities.

Figure 2.2 highlights a confusion that frequently arises when considering software creation and evolution activities. Software system life cycles, such as presented in Figure 2.2, tend to indicate a strong distinction between software development and software operation and maintenance whereas everyone's intuitive understanding of these activities is that operation and maintenance typically involves some development activity. Figure 2.2 indicates that this confusion comes from not clearly distinguishing the type of version being considered. Operation and maintenance of a software system may, and typically does, involve development of software releases and variants. In defining the details of software creation and evolution activities, one must, therefore, clearly recognize the type of version being addressed and allow for seemingly antithetical activities, performed in creating or evolving lower-level versions, to be naturally included as part of the defined activity.

2.4 Software Methodology

Many diverse processes and activities occur during a life cycle. These activities involve the accumulation of information and the representation of much of this information as software models. They also include the analysis of the models and information to assess the system's suitability as created or evolved up to some point in time.

A software method is a specific set of rules, techniques and guidelines for carrying out these processes and activities. Thus, a method serves to organize and discipline the overall process of preparing and evolving the software. Some software methods will be appropriate for software systems whereas others may only be appropriate for software variants. For example, to be useful for the preparation and evolution of a software system, a software method should address the problems of transitioning between releases.

A software methodology is a collection of methods. There are two interpretations of this definition. On one hand, the individual methods can be compatible ways of performing different activities -- for example, some can cover development activities and some can cover operation and maintenance activities -- with the result that the methodology itself is of broader life cycle coverage. On the other hand, all of the methods can cover the same set of activities, sharing some common aspects but differing in their details. In either case, a specification of the methodology

will identify those general principles, practices or procedures which are the basis for compatibility among the different methods or serve to highlight the commonalities among the similar methods.

Therefore, a software methodology is a general philosophy, or approach, for carrying out the software creation and evolution process. It provides general principles, practices and procedures for using software technology to prepare and evolve acceptable software. It also provides general principles, practices and procedures for using management technology to guide the overall process to a timely and cost-effective conclusion. Whereas the software methodology provides a general approach to the disciplined and systematic preparation and evolution of software, a software method translates this general philosophy into specific actions to be performed.

A software methodology is very closely related to the software environment providing the automated and manual tools supporting its use. These tools enforce or encourage following the methodology's guiding principles and using the methodology's practices and procedures. A very important connection between a methodology and an environment is that the methodology provides a basis for integrating the environment's tools. By having all of the tools support a single methodology, there may be a high degree of coherency stemming from the unifying conceptual basis provided by the methodology. The connection also extends in the other direction. It is possible to assemble a collection of tools without regard for the supported methodology. However, in such a case, the results will inevitably constrain the way in which software can be created and evolved. Therefore, such an environment will effectively impose its own, probably ad hoc, methodology.

2.5 Classes of Methodologies

A software method or methodology has many features, for example:

scope: extent to which it disciplines the creation and evolution of a software system rather than just the individual releases or variants,

potential for automated support: extent to which it can be supported by automated tools, and

coverage: extent to which it covers the full life cycle of some type of version.

Special names are sometimes used to denote methods or methodologies having specific scope or coverage features or

automated support potential. For example, a software system life cycle methodology covers a software system's full life cycle and an automated development method provides a set of automated tools for those activities preceding delivery of a software version.

Different types of methodologies can also be distinguished according to the general nature of the approach they specify. Thus there are general classes of methodologies, with all the methodologies in a class sharing some general features or characteristics. Some general classes of currently popular methodologies are discussed in the rest of this subsection.

2.5.1 Product-Oriented Methodologies

All methodologies result in the preparation of products and one approach to specifying a methodology is, therefore, to define these products rather than define the activities used to prepare and analyze the products. Most current methodologies are examples of this product-oriented class of methodologies and emphasize the intermediate and final products produced during the development activity.

A purely product-oriented methodology would specify the products but not the order in which they should be prepared. However, current instances of this class also divide the life cycle into phases, with the usual connotation that the phases are begun in a prescribed order and that the products from one phase should be finished and critically reviewed before progressing to the next phase. A typical set of phases for a software system is shown in Figure 2.3. The phases relate to the time points in the life cycle and indicate the general nature of what is done during the corresponding activities.

These methodologies generally do not specify particular activities that should be performed; rather they define the products that are to be produced and the general nature of how and when the products should be reviewed. This reflects the project management heritage of many product-oriented methodologies. The definition of the products and their review points is, in essence, a definition of milestones that provide management insight and control.

A typical product-oriented methodology is defined by DoD-STD-SDS for the development of DoD software systems (3). The general intent of this methodology is to wield management control over the acquisition of a software release. While the emphasis of the methodology is upon the initial release, the methodology is also of benefit for the competitive acquisition of subsequent

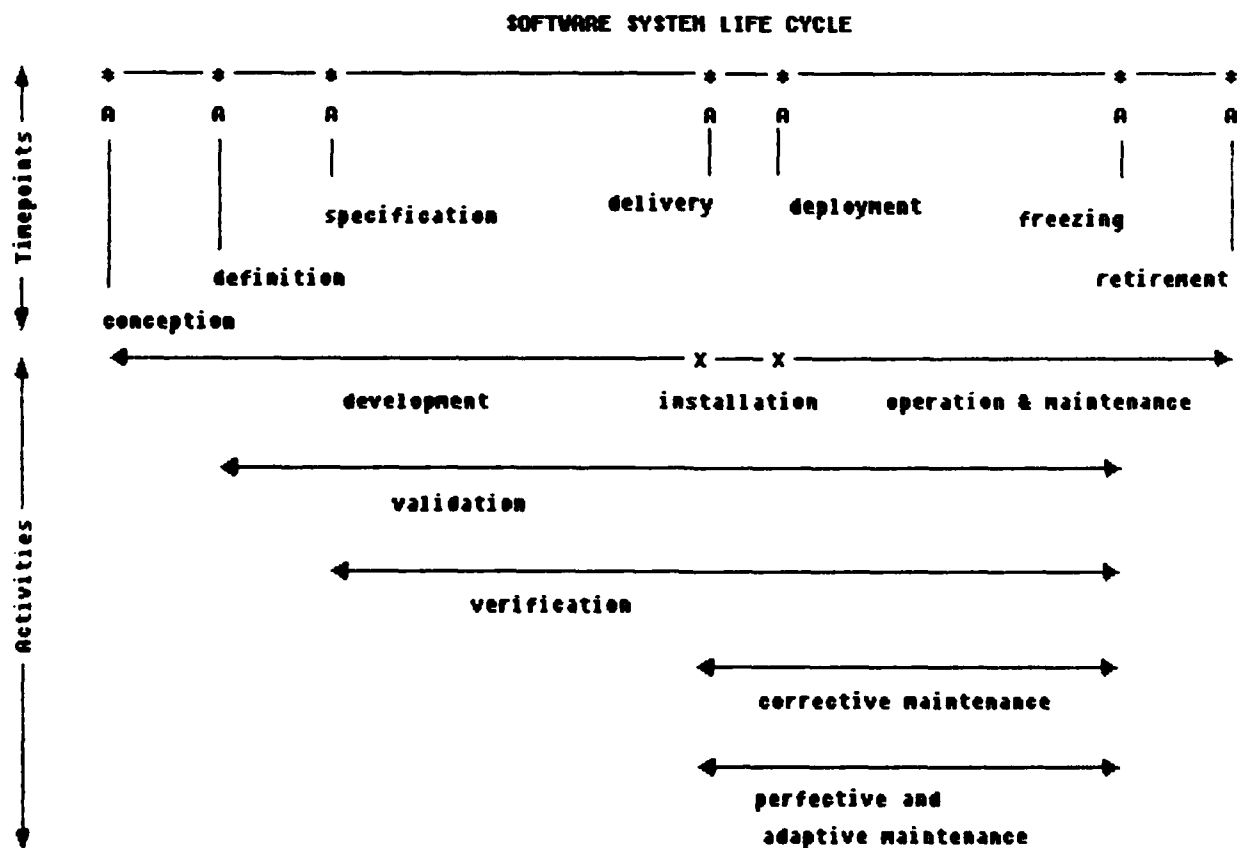
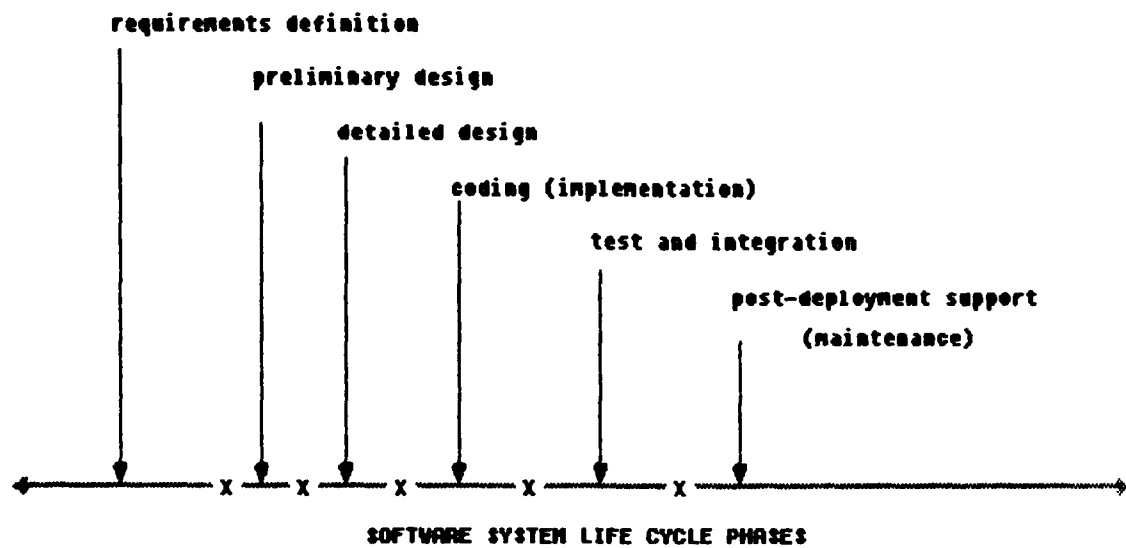


Figure 2.3: Relationship Between Software System Life Cycle Phases and Life Cycle Activities

releases. The methodology defined in DoD-STD-SDS focuses on (full-scale) development and defines the documents that are to be produced during the phases shown in Figure 2.3. It also defines some of the details of the activities during each phase, in particular the reviews that occur during and at the end of the phase. Finally, DoD-STD-SDS specifies several auxiliary activities such as configuration control and test management.

2.5.2 Other Methodology Classes

Since the idea of organizing the life cycle into phases first appeared in the mid-1960's, several product-oriented methodologies have evolved and matured. While they have proven to be generally useful, some problems have been experienced. For example, where there is little experience with the application being addressed, a totally logical progression from high-level, application-specific concerns to low-level, implementation concerns has proven to be difficult. In some cases, methodologies that are more attuned to the particular application have been found to be more effective. Also, it has sometimes been difficult to incorporate new technology into product-oriented approaches to development. These problems have led to the introduction of other types of methodologies which, while not yet as well developed, have been receiving a fair amount of attention. These alternatives of course result in the preparation of products, but focus more attention on the techniques used to carry out the process.

Data structuring methodologies are one example of these newer methodology classes. Product-oriented methodologies tend to emphasize the top-down decomposition of a system's functionality. However, data structuring methodologies focus attention on the structure of the data being processed and the composition of processing steps that perform the system's intended data transformations.

Another class is object-oriented methodologies. Product-oriented methodologies tend to emphasize development of the software system's functionality and, therefore, are primarily concerned with modeling real-world operations in the software. The object-oriented approach, on the other hand, provides a more balanced treatment of objects and operations. It attempts to mirror the problem space in the solution space by identifying the (data and non-data) objects of interest and the operations which act on these objects. The organization and operation of the software is described by modeling the interactions among objects. The software can be developed either using composition or decomposition techniques.

Another class of methodologies are prototyping methodologies. Prototyping has been successfully used in other disciplines, such as engineering and architecture, as a way of coping with risk. It provides "rough cuts" useful in determining

the feasibility and suitability of emerging solutions. To date, software prototyping has primarily been used to support product-oriented methodologies as a way of clarifying the software requirements. Through the prototype, users can get a feel for whether the system will satisfy their desires and expectations. In turn, the developers can extend their understanding of what the users really need.

Prototyping methodologies result from using prototyping as the overall approach rather than just in support of requirements definition. With such a methodology, an immature version is initially built and then gradually elaborated to provide the full, required functionality. In many ways, such a methodology reflects what naturally occurs in software projects.

2.5.3 Mixtures of Methodologies

The different methodologies discussed above have different emphases. Therefore, it could prove valuable to use different ones for different aspects of software creation and evolution. At the moment, there is little experience in combining methodologies. As more experience is gained, methodology combinations can be expected to become more prevalent.

As an example, a recent empirical study (4) has shown that prototyping is valuable when experience in the application area is lacking and the risk in being able to develop a suitable piece of software is high. The same study indicates that product-oriented methodologies are valuable in producing production-quality systems. This suggests that it might be beneficial to use a prototyping methodology early in a project and then switch to a product-oriented methodology for later releases.

Different methodologies could also be used for the different levels of concern discussed previously. For example, a prototyping methodology could be used to produce software variants, whereas a product-oriented methodology could be used to guide the overall process at the software system level.

2.6 Summary

Many approaches are available for coping with the risk, uncertainty and complexity associated with most software projects. The majority of these methodologies attempt to provide the needed discipline by emphasizing the gradual accumulation of information about the software's operational detail. Others emphasize the use of specific modeling techniques to capture the information in unambiguous, rigorous terms. Few specify concrete techniques for analyzing the suitability of this information, other than late in the project when the software is close to being operational.

The majority of current methodologies specify a set of work-products to be successively (and perhaps iteratively) developed. There are several techniques (such as prototyping) and partial life cycle methods (such as object-oriented programming) supporting this product-oriented approach. Concrete methods must be defined to capture these techniques in a usable form.

A complete methodology must address three major categories of concern: those associated with assuring that the software fulfills its role in the overall automated system, those associated with preparing software that meets its specification, and those associated with upgrading the software to meet changing requirements. No current methodologies address all of these concerns. It is most likely that a methodology which addresses all levels of concern will be a composite with different methodologies being used to address different concerns. Preparing such composite methodologies will require a way of determining the differences and similarities among methods. Methodologies themselves are a way of highlighting differences and similarities since the definition of a methodology provides an accounting of the features common to all of the methods encompassed by the methodology. By defining methodology classes, this highlighting can be emphasized even further.

Concrete definitions of methods, methodologies and methodology classes are prerequisite to determining the compatibility information needed to define composite methodologies. However, a concrete definition requires the identification of characteristics useful in making the distinctions necessary to set various methods, methodologies or methodology classes apart from each other.

An appropriate set of characteristics can also provide a basis for classifying and evaluating methodologies as well as selecting among alternative methodologies. The general nature of these technologies is discussed in the next section. This serves to further define the nature of the characteristics which are needed prior to a discussion, in Section 4, of the characteristics themselves.

3.0 CLASSIFICATION, EVALUATION AND SELECTION

A means to classify, evaluate or select methodologies could be used to support many activities:

- identifying methodologies for use on specific software projects,
- identifying methodologies which need further development to support some particular purpose, such as maintenance in particular application areas,
- developing a "consumers guide" to methodologies,
- choosing methodologies for empirical studies into issues such as how to provide effective automated support, and
- qualifying a methodology with respect to a set of requirements.

The general nature of methodology classification, evaluation and selection technologies, and the process of developing them, are discussed in this section. The discussion here concerns their use in identifying methodologies for specific projects. This orientation has the benefit of highlighting inter-relationships and interdependencies of the technologies; it is not meant to imply that this would be their only use.

3.1 Overview

The three technologies and their primary inter-relationships are pictured in Figure 3.1. (In this section's figures, processes and activities are indicated by names with all capital letters whereas information and knowledge are indicated by lower case names.) The overall process is to consider all possible methodologies and identify those acceptable with respect to the criteria associated with a particular purpose. The major activity is therefore selection of the acceptable methodologies. This selection requires two supporting activity: an evaluation activity which determines the "value" of methodologies and a classification activity which determines a methodology's class. The relationship among these activities is discussed in subsequent sections.

Overall guidance is provided by criteria, that is, by specifications of need in terms of the methodology user's desires. These criteria are purpose and project specific and for a project reflect the target software's application area, the project's management structure, the policies of the contractor and contracting organizations, and the basic nature of the development and post-deployment support techniques and tools used in the project. Ideally, the classification, evaluation and selection

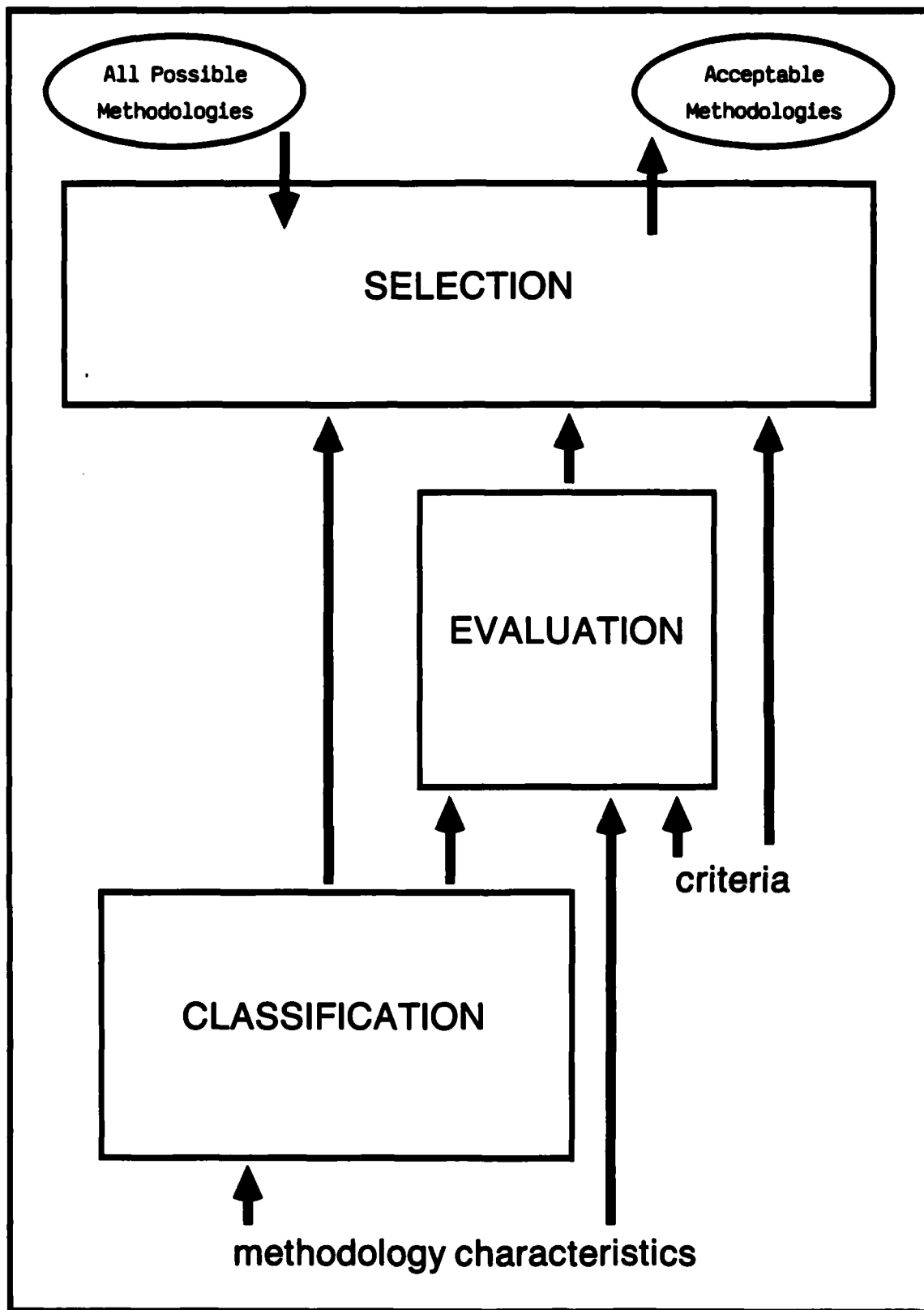


Figure 3.1: Selection, Evaluation and Classification Technologies

technologies will identify a single, "best" methodology. But more typically, it will identify a collection of acceptable methodologies and additional criteria (or subjective analysis) may have to be used to select a single methodology from among the acceptable ones. Overall support is provided by characteristics, that is, by attributes of methodologies useful for comparing and contrasting them. These characteristics will be discussed later.

3.2 Focusing on Ada-Compatible Methodologies

Rarely are all possible methodologies considered. Rather, requirements serve to reduce the set of possibilities and provide an initial focus. Since our interest is primarily with methodologies supportive of the use of Ada, the set of requirements in this case serve to focus upon those methodologies that can be efficiently and effectively used for Ada-based software systems. This is indicated in Figure 3.2.

This focus limits the scope of the classification, evaluation and selection technologies that are produced. The various activities need only be able to handle selection from among Ada-compatible methodologies. The various pieces of knowledge need only pertain to these methodologies. While the technologies may be useful in considering other methodologies, this would be a pleasing side-effect rather than a specific goal.

3.3 Selection Technology

The activity of selection identifies those methodologies meeting the criteria specified for a particular project. Selection can be performed in many different ways, but each will generally involve focusing on a set of candidate methodologies and choosing among the candidate methodologies. These subactivities are highlighted in Figure 3.3.

Focusing narrows in on a set of methodologies to be given detailed consideration. This narrowing of attention could be done at any point in the process of identifying acceptable methodologies. It could even be done more than once, interleaved with the activity of choosing among the methodologies that previous activities have identified as potential candidates.

Focusing could result from consideration of the criteria. For example, the criteria might specify or imply that only prototyping methodologies should be considered. Alternatively, focusing could follow from consideration of factors that are not reflected in the criteria. For example, an organization might prepare a list of "approved" methodologies and the choice is limited to methodologies from this list. Therefore, focusing may require a knowledge of methodology classes and a means of determining whether or not a

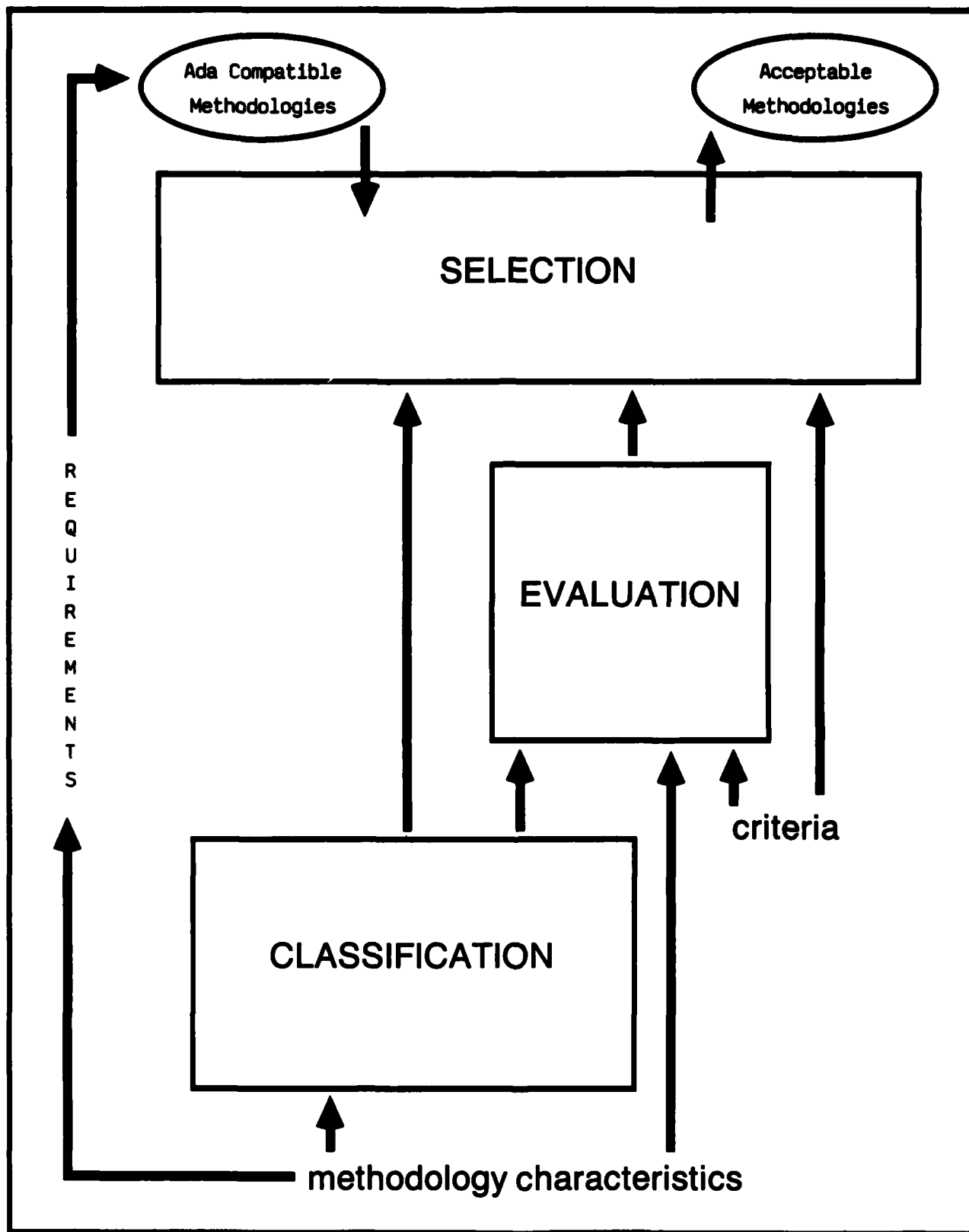


Figure 3.2: Selection, Evaluation and Classification for Ada-compatible Methodologies

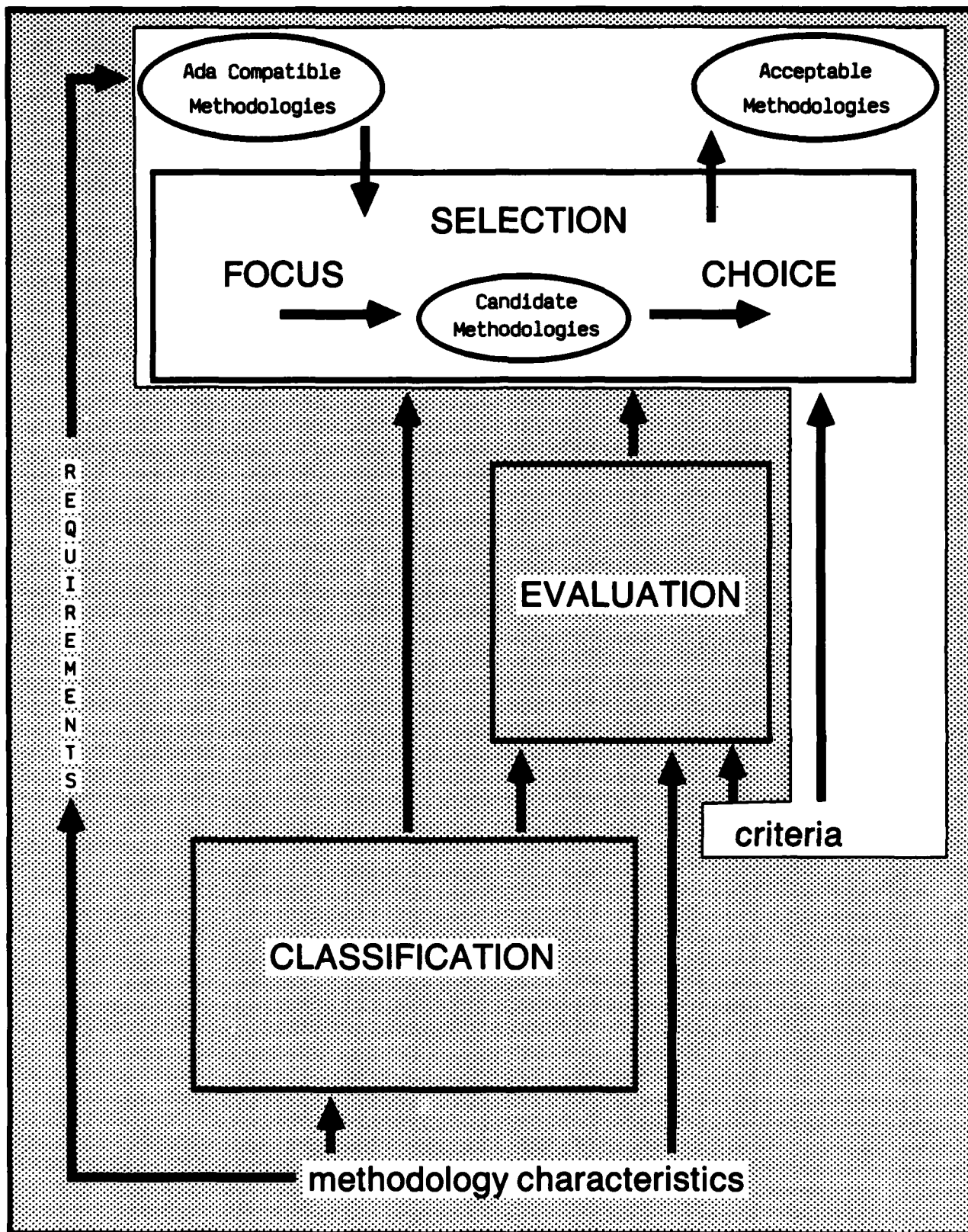


Figure 3.3: Selection Technology
3-5

specific methodology is in a particular methodology class. It may also require the ability to evaluate an entire methodology class without evaluating each individual methodology in the class.

Subsequently, choosing among the candidate methodologies involves evaluation with respect to the criteria and interpretation of the results to identify acceptable methodologies. Again, many approaches are possible, but each will require the ability to determine the potential value of a methodology, or class of methodologies, with respect to the criteria.

3.4 Evaluation Technology

Selection requires the ability to determine how well individual methodologies meet the criteria. This determination leads to a need for measures, that is, quantitative attributes that can be used to determine whether or not a methodology meets the criteria. These measures can be obtained by identifying useful, existing measures or defining new ones through some measure identification and definition activity. Once the measures have been identified or defined, they can be used as part of an evaluation activity that results in the methodology evaluations needed to carry out selection. These new pieces of technology are highlighted in Figure 3.4.

Sometimes, the criteria may be specific and quantitative enough to be used directly as measures. More often, the criteria, being use and user oriented, will only imply the measures to be used. For example, "average coupling among modules" is a measure that could be used to determine whether or not a methodology meets the criteria "methodology should enhance maintainability." Prior work and experience may already have led to a set of measures pertaining to the criteria. Often, however, new measures will have to be defined that are specific to the task of evaluating a methodology with respect to the stated criteria.

Exactly which measures to use in any particular evaluation activity may depend on the class of the methodology being evaluated as well as on the criteria. Thus, the measure identification and definition activity will, in general, provide a set of measures to be conditionally used depending on the class of the methodology being evaluated.

The definition of meaningful, useful measures has proven notoriously difficult and most software measures defined to date concern only a system's final implementation. Effective methodology evaluation will require measures that pertain to other products, as well as to the development or post-deployment support process itself, the developers of the methodology, and the qualifications needed to effectively use the methodology. Preparation of a set of pre-defined measures and the development of a measure identification and definition procedure is, therefore,

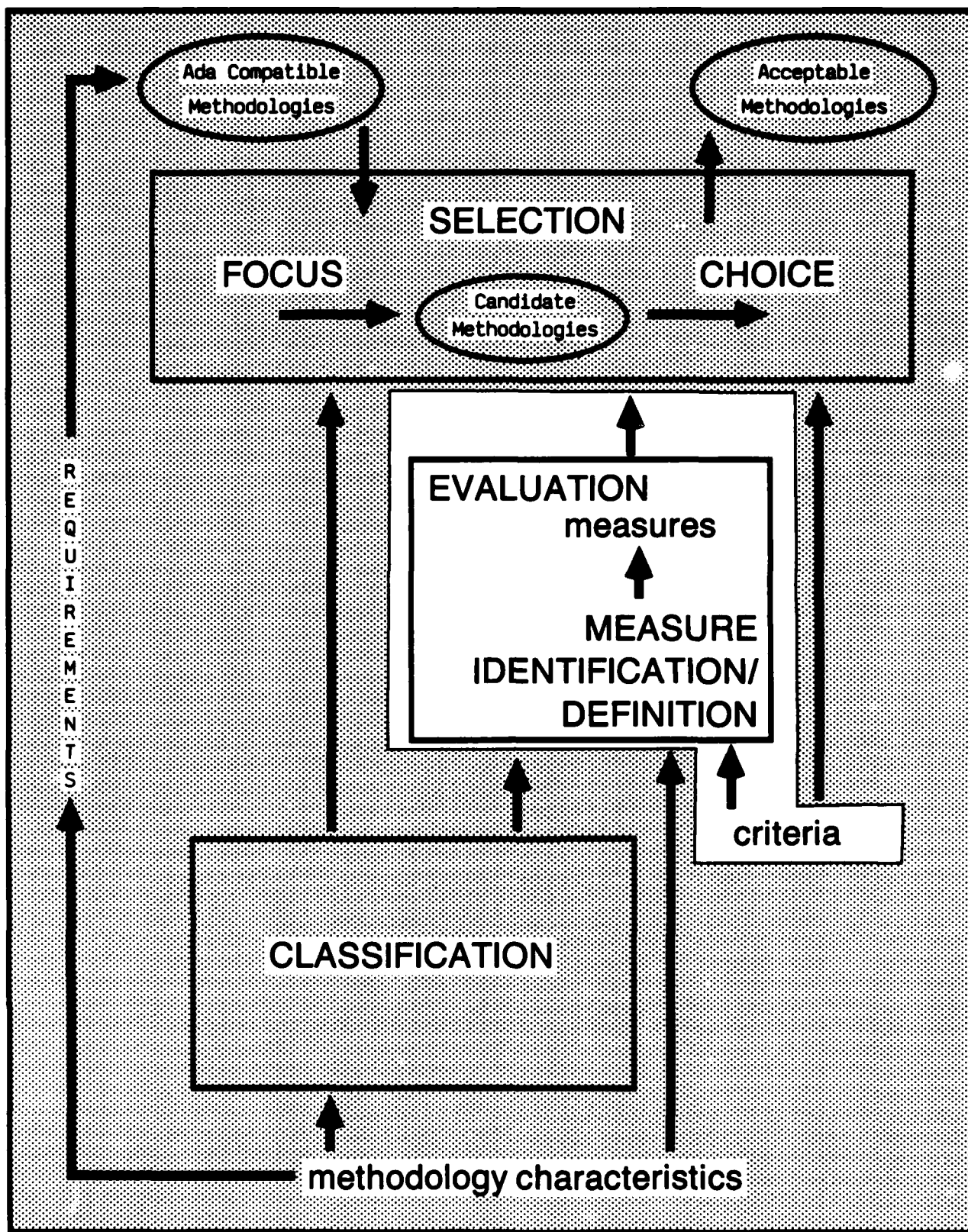


Figure 3.4: Evaluation Technology

critical. (The STARS program has started to address this critical need through the preparation of an initial set of measures (5).)

Once measures have been identified and defined, evaluation can be performed. The basic activity during evaluation will be to determine each methodology's "value" with respect to the criteria. Since the course of evaluation may depend on the class of the methodology under evaluation, information about the various classes may have to be available during evaluation. This methodology class information may also allow evaluation to proceed more efficiently since it might be possible to evaluate entire classes of methodologies as a whole.

3.5 Classification Technology

The evaluation and selection of methodologies requires a knowledge of methodology classes and a classification activity allowing determination of a methodology's class. It also requires a knowledge of methodology characteristics that can be used as a basis for quantitative measures. These final pieces of technology, along with the class definition activity needed to develop the required knowledge of methodology classes, are highlighted in Figure 3.5.

Methodology characteristics support the definition of both evaluation measures and methodology classes. As with measures, these characteristics may merely be attributes of the products produced by the methodology. Or, they may be attributes of the methodology itself, the developers of the methodology, or the users of the methodology. For example, the following attributes could be used in characterizing a methodology: "average number of software modules on a critical timing path", "proportion of real-world efficiency constraints traceable to units in software products", "number of years of experience of methodology's developers in the area of flight control software", and "required experience in application area needed to make effective use of the methodology". These examples indicate that characteristics can be relatively ill-defined, such as the last example, or relatively well-defined, such as the third example. To be useful in defining measures of methodology types, ill-defined characteristics will have to be elaborated in terms of other, more well-defined ones.

Additionally, classification requires that it is possible to objectively determine the characteristics of a methodology. Thus, if a characteristic is not stated in measurable terms, it will be necessary to identify the metrics pertinent to assessing the characteristic and define the characteristic in terms of these metrics.

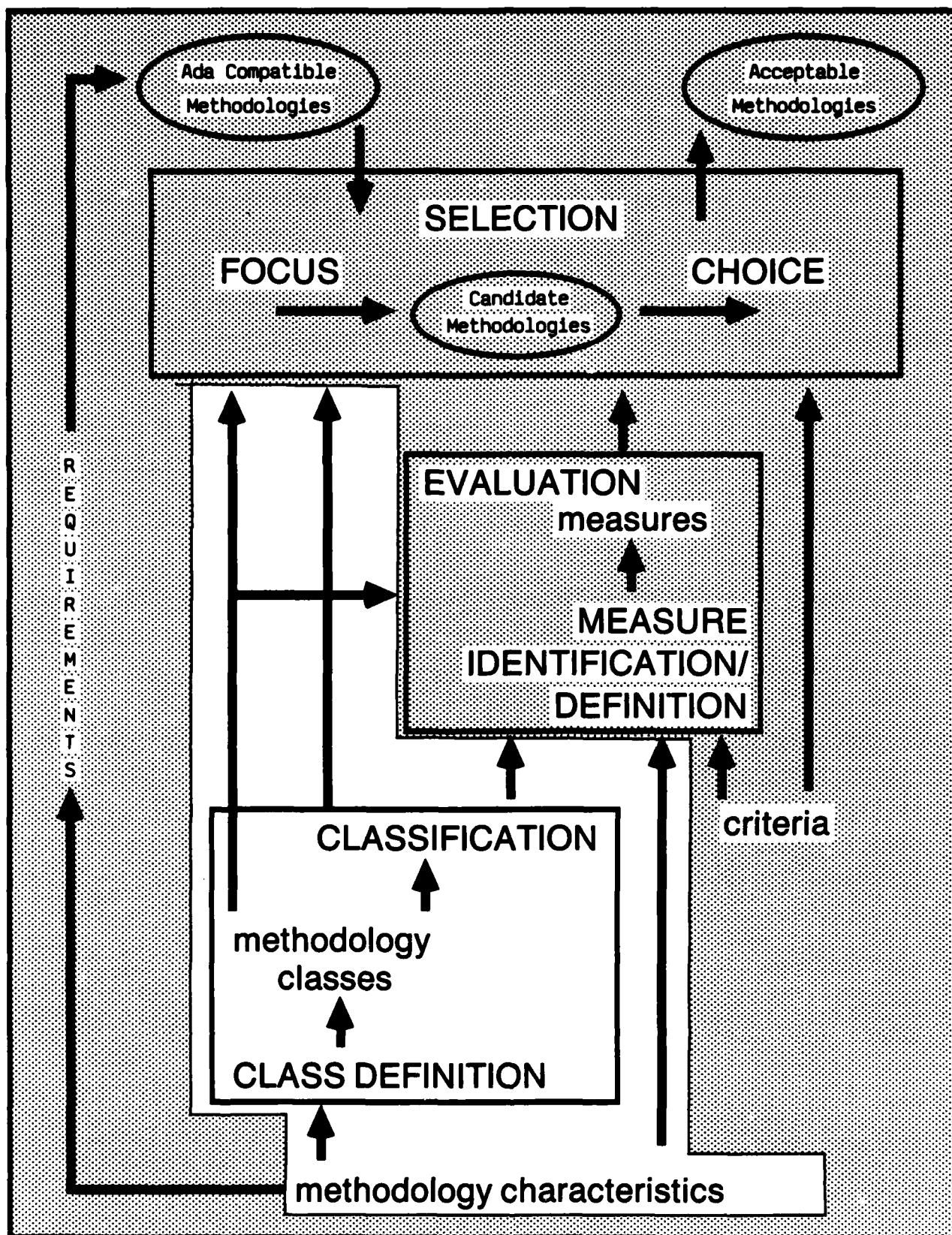


Figure 3.5: Classification Technology
3-9

Criteria, measures and metrics are seemingly similar terms that are used here in specific ways, as defined in the preceding discussion. To recap the distinctions which have been made: criteria reflect goals or requirements; whether or not a methodology meets a criteria is determined by "computing" various measures and then interpreting the values obtained; and a measure is "computed" by determining a methodology's characteristics and this is usually most easily done by computing specific, well-defined metrics. Using an analogy to selecting a car, the criteria "economical" is determined in terms of the measures "fuel consumption", among others, and determining this measure requires computing the metrics "mpg, city" and "mpg, highway".

Characteristics may be used to define methodology classes by defining the attributes common to all methodologies in the class. The characteristics may be determined by identifying some broad type of methodologies (such as prototyping methodologies) and then determining the low-level, well-defined attributes which characterize methodologies of this type. The set of characteristics can therefore be incrementally built up by considering different classes in turn and adding any pertinent characteristics found to be missing from the set.

The classification activity could be done once in preparation for evaluation or selection, with periodic updates. Or it could be done as the results are needed. Whenever it is done, it involves using the characteristics to determine the class or classes to which a methodology belongs. It leads the person performing the classification through a series of questions, the answers to which determine the characteristics of the methodology and eventually lead to the identification of the methodology's classification.

Like all the activities discussed above, the classification activity may be empirical. In many cases, the only way to classify (or evaluate or select) a methodology may be by experimentally using it in some trial situations. Ideally, this experimental usage will be limited and involve investigation of only "small" systems. However, the state-of-the-art in experimental methodology evaluation is currently insufficient to guarantee this.

3.6 Development of the Technology

The preceding discussion of classification, evaluation and selection has introduced the various pieces of technology in a logical order and uncovered their more obvious interdependencies. However, it is unlikely that development of this technology will be able to progress "top-down" in the order presented above. A major reason is that it will undoubtedly prove valuable to prototype the full technology in order to understand how to best evolve it into something more extensive and useful.

Another major reason is that the classification, evaluation and selection technologies will likely be empirical in nature, and each will be needed to support the others. For example, the definition of methodology classes may require the ability to select a methodology to experiment with or the ability to evaluate a methodology with respect to some measures. This leads to the secondary dependencies shown in Figure 3.6.

Thus, the technology must be gradually and iteratively elaborated over time. Each step in this elaboration involves:

- define characteristics: define a set of characteristics that supports the classification, evaluation and selection of methodologies; typically this will be the set that results from previous technology elaboration steps;
- develop a classification technology: develop a class definition procedure and use it to define methodology classes; develop a methodology classification procedure; use previously developed evaluation and selection technology as necessary; expand the set of methodology characteristics as needed;
- develop an evaluation technology: define quantitative measures and the means to define selection criteria as a set of measures to be evaluated; provide the ability to determine a methodology's "value" with respect to these measures; use previously developed classification and selection technologies as necessary; expand the set of methodology characteristics as needed; and
- develop a selection technology: develop procedures for focusing upon candidate methodologies and choosing among a candidate set of methodologies; use previously developed classification and evaluation technologies as necessary; expand the set of methodology characteristics as needed.

In parallel with these activities, the requirements for Ada-compatible methodologies will be continuously under refinement. The refinement will result from the knowledge gained in developing the classification, evaluation and selection technologies. It will also affect the development of these technologies in that it will focus work on methodologies applicable to the development and post-deployment support of Ada-used software systems.

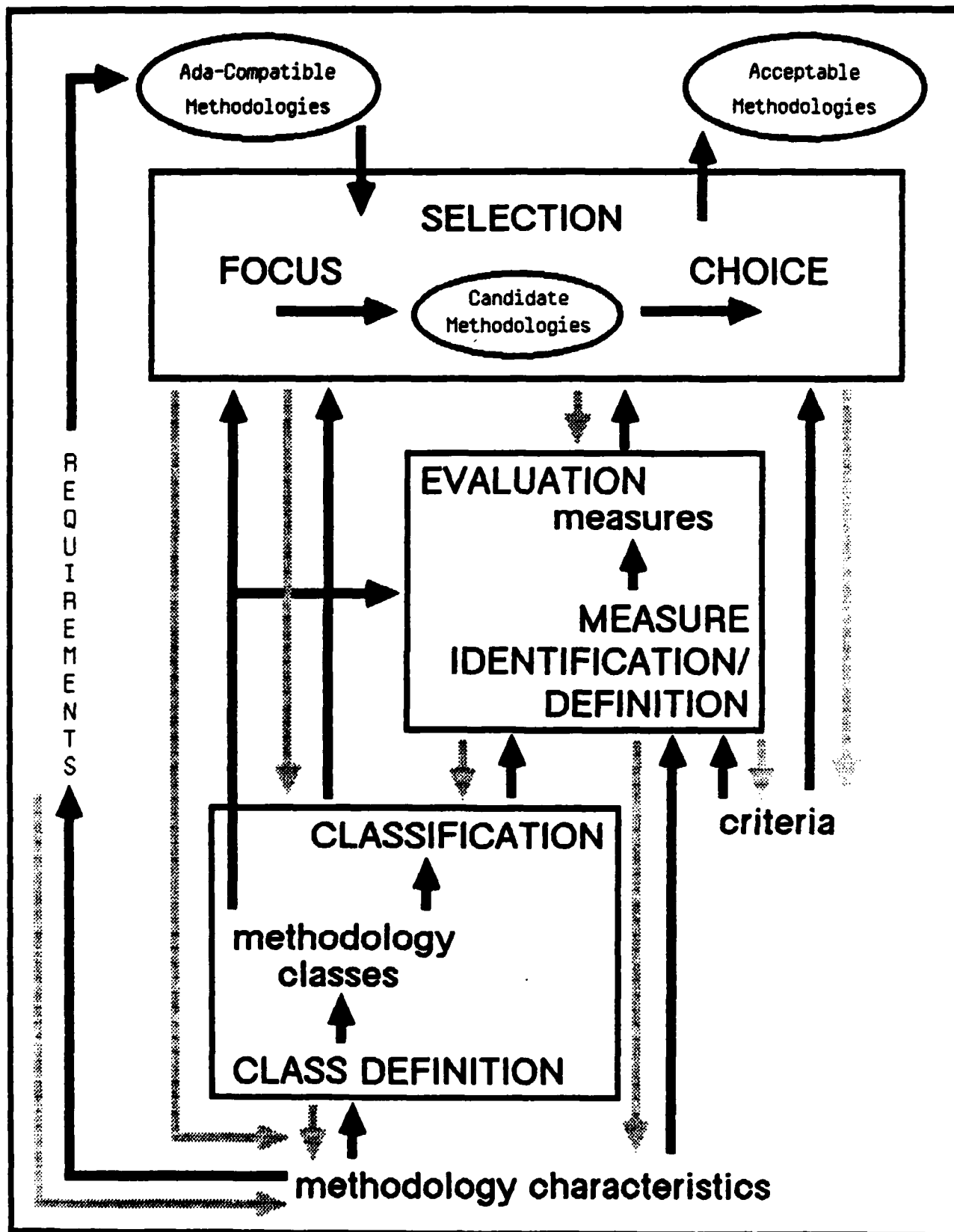


Figure 3.6: Classification, Evaluation and Selection Technologies for Ada-compatible Methodologies Including Secondary Dependencies

While it may appear most logical to progress through the above activities in the order in which they are presented, this is not necessary. In fact, not all of them need to be completed for every step in the technology elaboration. Each elaboration step will result in a better understanding, not only of the emerging technology, but also of the activities that are needed to mature it further. Thus, each elaboration step will provide guidance on what to do at the next step.

Critical to all of the technology discussed here is a set of characteristics which will constantly undergo expansion as the technology is developed. A framework is needed to structure this set so that the expansion can be conducted in an orderly manner. These characteristics, and their organization via a framework, are discussed in the next section.

4.0 METHODOLOGY CHARACTERISTICS FRAMEWORK

Methodology characteristics are central to the development of selection, evaluation and classification technologies. Potentially, there is a very large number of characteristics useful for describing, comparing and contrasting methodologies. The set of characteristics may potentially never be complete but continue to grow to reflect the appearance of new criteria or the development of new methodologies. Consequently, there must be an organizing framework for the characteristics that can be extended as necessary and easily accommodate new characteristics as they are uncovered.

A preliminary framework for organizing methodology characteristics and a procedure for enumerating an initial set of characteristics are introduced in this section. The intent to date has been to determine the general nature of the framework, systematize the enumeration of characteristics and populate the framework with some example characteristics sufficient to demonstrate its effectiveness and rationale. In the near future, the framework will be used for some initial classification activities. This will support validating and further refining both the framework and its underlying conceptual basis.

4.1 Overall Structure of the Characteristics Framework

The characteristics framework provides a gross organization for methodology characteristics by defining four major categories, indicated in Figure 4.1. These categories highlight four major concerns which arise when considering methodologies for use on Ada-based software projects. As such, they separate the characteristics along the lines of the types of criteria that are likely to be encountered.

A fifth category, automated support, was suggested by the previous work done by Freeman and Wasserman (1). It was not preserved because it was felt that this concern most naturally cuts across all categories. In the framework described here, characteristics of the automated tools provided by a methodology are categorized according to the nature of the support provided by these tools. As a consequence, the framework makes a strong association between the characteristics of a methodology and the characteristics of the automated tools supporting use of the methodology.

Subcategories of characteristics exist for each of the major categories. The emphasis so far has been on determining the subcategories within the technical category. These particular subcategories have been delineated by considering the different types of software versions and various approaches for producing them. As discussed in Section 2, there are three major types of

Definition

Technical	Characteristics concerning how the methodology supports the preparation of products having such desirable properties as modifiability, efficiency, reliability, understandability and reusability; includes characteristics of the production process (such as resource consumption) as well as characteristics of the produced products.
Management	Characteristics concerning the support which the methodology provides for management activities such as planning, tracking, resource allocation and cost estimation.
Usage	Characteristics concerning the process of acquiring and using the methodology; including activities such as purchase, installation of automated support, usage monitoring, customization, extension and training.
Ada-Compatibility	Characteristics concerning the methodology's encouragement of effective use of the Ada language and its underlying concepts.

Figure 4.1: Definition of Characteristics Categories

software versions - software systems, software releases and software variants - and these help in distinguishing among various methodological concerns. Focusing on these types of versions and approaches for producing them provides the further categorization depicted in Figure 4.2.

This figure portrays the gross structure of an extensible framework for methodology characteristics. This pictorial representation emphasizes that by focusing attention down through the levels of categorization, one arrives at a collection of characteristics that is pertinent to that focus. However, the strict partitioning of characteristics implied by the figure is not likely to exist in practice. It is expected that an individual characteristic will be pertinent to many different focuses and will therefore appear in several collections of characteristics. Thus, the framework should not be viewed as a mechanism for partitioning the characteristics. Rather, it provides a means for categorizing the characteristics and, as such, assists in identifying sets of characteristics that have a particular emphasis or utility.

Figure 4.2 also emphasizes that the part of the framework receiving the most attention concerns product-oriented approaches to producing variants. Some attention has been given to the other three major categories and this is discussed later in this section.

Finally, the figure indicates the extensibility of the framework to accommodate new approaches. Potentially, new subcategories could be added anywhere in the structure. Nonetheless, the expectation is that the first two levels will be relatively static and that future extensions will lead to additions to the collections of characteristics, the incorporation of new sub-subcategories within the technical category for additional methodology classes, and expansion at the subcategory level for the other major categories.

4.2 Lower Level Structure of the Characteristics Framework

The overall structure of the characteristics framework serves to identify collections of characteristics that are related in terms of their emphasis on various concerns or issues. However, further detailed organization of the individual collections is needed for three reasons. First, it supports understanding how the characteristics relate to each other. Second, it is needed to understand how they can be used in comparing and contrasting methodologies. Finally, the organization can help in assuring that the collection is reasonably complete. This subsection discusses an approach to organizing collections of characteristics serving these purposes.

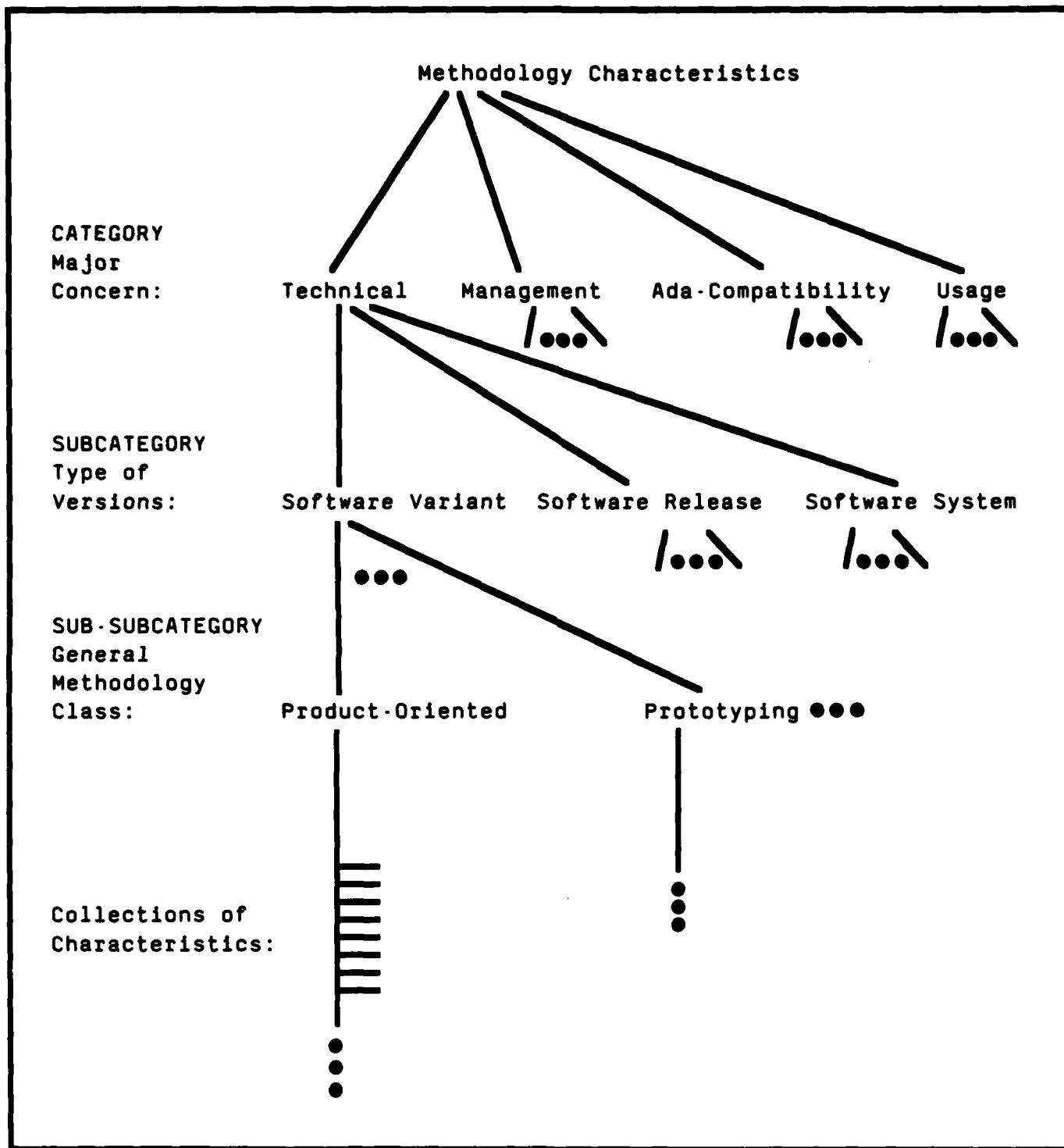


Figure 4.2: Characteristics Framework

Collections of technical characteristics may be organized using a two dimensional matrix. This matrix is based on the work reported by Ross, Goodenough, and Irvine (6), and is shown in Figure 4.3. The first dimension concerns product quality goals such as correctness, reliability and portability. The second concerns general principles that have emerged as beneficial in meeting these goals. These two dimensions allow organizing a list of characteristics according to how they reflect a methodology's support for following one or more of the principles to achieve one or more of the goals.

The matrix can be used to structure any of the collections of characteristics found under the major category of technical characteristics. The definitions for the goals and principles, given in Figure 4.3, are in terms of software variants but can easily be changed to pertain to software releases and software systems. Also, the principles and goals are common across many approaches. Other matrices will be needed for the other three major categories since different concerns and issues are of importance for these categories.

A possible alternative use for these matrices is to check the completeness of a set of characteristics, or their associated metrics. For example, after inserting the characteristics developed by RADDC (7) into the technical matrix, one could check their completeness with respect to the goals and principles of software engineering as regards product-oriented methodologies and technical concerns. As other matrices are developed for other classes of methodologies and other broad categories of concerns, they could be similarly used to check completeness in these other situations.

4.3 Populating the Framework with Characteristics

Our early attempts to enumerate technical characteristics failed because of the lack of a concrete focus. At that time, deciding whether a particular characteristic was pertinent and how it related to other characteristics was largely an intuitive matter. Also, there was a lack of consistency in the definition and level of detail among the identified characteristics.

Experience proved that three things were necessary to support the effort of enumerating characteristics. First, it was useful to narrow attention to a specific major concern (such as technical issues), a specific type of software version (such as software variants), and a specific class of approaches for developing the version (such as product-oriented approaches). Second, it was useful to consider methodology characteristics in terms of how the methodology supports the use of modern software technology principles (like information hiding) to achieve results with

COUPLE one software variant with another.
 INTEROPERABILITY: Effort required to variant of the functions that the software performs.
 scope of the functions that the software variant can be used in other applications; or related to the packaging and variant to which a software system to another.
 REUSABILITY: Extent to which a software variant can be used in other applications; or related to the packaging and variant to which a software system to another.
 PORTABILITY: Effort required to transfer a software variant from one hardware configuration and/or software system to another.
 FLEXIBILITY: Effort required to modify an operational software variant.
 TESTABILITY: Effort required to test a software variant to insure it performs intended function.
 MAINTAINABILITY: Effort required to locate and fix an error in an operational software variant.
 USABILITY: Effort required to learn, operate, prepare input, and interpret output of a software variant.
 INTEGRITY: Effort required to learn, operate, prepare input, and interpret output of a software variant.
 EFFICIENCY: The amount of computing resources and code required by a software variant to perform a function.
 RELIABILITY: Extent to which a software variant can be expected to perform its intended function with required precision.
 CORRECTNESS: Extent to which a software variant satisfies its specification and fulfills the user's mission objectives.
 GOALS
 PRINCIPLES

MODULARITY: Assists structuring a software variant.
 ABSTRACTION: Assists identifying essential properties common to superficially different entities.
 LOCALIZATION: Assists bringing related things into physical proximity.
 HIDING: Assists making inessential information inaccessible.
 UNIFORMITY: Assists ensuring consistency and freedom from unnecessary differences.
 COMPLETENESS: Assists ensuring that nothing essential is omitted.
 CONFIRMABILITY: Assists ensuring that information needed to verify correctness has been explicitly stated.

Figure 4.3: Preliminary Matrix for Collections of Technical Characteristics

various quality attributes (such as reliability). Third, it was helpful to think in terms of various objectives that serve to meet quality-related goals but rely on, or relate to, one or more of the modern software technology principles.

The first two experiences led to the framework's gross structure and the characteristic organizing matrices, respectively. The third led to developing an approach for enumerating characteristics to populate the various cells in the matrices. It must be emphasized that this is only one possible enumeration approach which has been found particularly helpful in initially populating the matrices with characteristics. Other approaches -- such as scanning independently developed lists of characteristics or taking note of characteristics during demonstrations or experiments -- may prove more useful in the future when the task will shift to expanding the set of characteristics rather than determining an initial set.

The enumeration approach consists of considering each goal in turn, determining the objectives that support meeting this goal, converting these objectives into characteristics and then placing these characteristics into the cells corresponding to the principles that support meeting the identified objective. The first two columns of Figure 4.4 show the results of carrying out the first three of these steps for technical characteristics pertinent to the goal of efficiency and the production of software variants. The right-hand column of Figure 4.4 indicates which cells in the efficiency column would receive the enumerated characteristics when they are placed in the technical matrix.

While these characteristics were enumerated within the context of product-oriented methodologies, the result can be expected to be pertinent to most other methodology classes. This re-emphasizes two previously made points. First, the matrix for technical characteristics is relatively independent of the methodology being considered since the matrix is founded on the goals and principles which must be achieved and utilized, respectively, by any methodology. Second, the enumeration approach is useful for determining an initial set of characteristics and other approaches may be more useful for expanding this initial set.

While use of this enumeration approach is primarily for determining an initial set of characteristics, the focus on objectives does have two beneficial side-effects which may expand its utility. First, since objectives tend to imply some measurement, characteristics which are derived from objectives are themselves likely to be measurable. Additionally, the eventual use of the characteristics framework in selection activities and for providing a terminology for stating requirements of methodologies makes this orientation towards objectives potentially doubly useful.

OBJECTIVES	CHARACTERISTICS	PRINCIPLES
<p>01: Promote attaining an overall efficient software structure.</p> <p>01.1: Increase evaluation of how alternative software structures satisfy the efficiency constraints.</p> <p>01.2: Increase investigation of trade-offs between particular efficiency constraints which will increase overall efficiency.</p>	<p>C1: Extent to which software structure supports achieving efficiency constraints.</p> <p>C1.1: Number of alternative software structures evaluated.</p> <p>C1.2: Ratio of number of trade-offs considered against number of efficiency constraints.</p>	<p>Modularity. Localization</p>
<p>02: Increase ease of identifying those units in a software product which are affected by an efficiency constraint.</p> <p>02.1: Increase traceability between real world efficiency constraints and units in software products.</p> <p>02.2: Minimize complexity of software structures.</p>	<p>C2: Ease of identifying software units which are affected by efficiency constraints.</p> <p>C2.1: Proportion of real world efficiency constraints which are traceable to units in software products.</p> <p>C2.2: Degree of software structure complexity.</p>	<p>Modularity. Abstraction</p>
<p>03: Highlight information pertinent to critical throughput demands.</p> <p>03.1: Reduce number of software units to consider for a throughput demand.</p> <p>03.2: Reduce amount of documentation to consider for a throughput demand.</p>	<p>C3: Amount of information that needs to be considered for critical timing constraints.</p> <p>C3.1: Average proportion of software units investigated for each timing constraint.</p> <p>C3.2: Average number of documentation pages referenced for each timing constraint.</p>	<p>Abstraction, Hiding</p>

Figure 4.4: Technical Characteristics Pertaining to Efficiency and Product-oriented Methodologies

OBJECTIVES	CHARACTERISTICS	PRINCIPLES
<p>D4: Highlight information pertinent to storage constraints.</p> <p>D4.1: Reduce number of software units to consider for a storage constraint.</p> <p>D4.2: Reduce amount of documentation to consider for a storage constraint.</p>	<p>C4: Amount of information that needs to be considered for storage constraints.</p> <p>C4.1: Average proportion of software units investigated for each storage constraint.</p> <p>C4.2: Average number of documentation pages referenced for each storage constraint.</p>	<p>Abstraction. Hiding</p>
<p>D5: Minimize the scope of a software product affected by timing constraints.</p> <p>D5.1: Reduce the number of software units on a critical timing path.</p>	<p>C5: Extent of scope of a software product which is affected by timing constraints.</p> <p>C5.1: Average proportion of software units on a critical timing path.</p>	<p>Hiding. Localization</p>
<p>D6: Minimize the scope of a software product affected by storage constraints.</p> <p>D6.1: Reduce the number of software units which access data subject to a storage constraint.</p>	<p>C6: Extent of scope of a software product which is affected by storage constraints.</p> <p>C6.1: Average proportion of software units which access data subject to a storage constraint.</p>	<p>Hiding. Localization</p>

Figure 4.4: continued

OBJECTIVES	CHARACTERISTICS	PRINCIPLES
<p>07: Ensure complete identification of efficiency constraints.</p> <p>07.1: Increase cross-referencing of efficiency constraints to particular software units.</p> <p>07.2: Increase early identification of all efficiency constraints.</p>	<p>C7: Extent of identification of efficiency constraints.</p> <p>C7.1: Proportion of efficiency constraints which can be traced to software units.</p> <p>C7.2: Number of efficiency constraints not initially identified.</p>	Completeness
<p>08: Ensure complete documentation of all efficiency constraints.</p> <p>08.1: Increase specification of all efficiency requirements.</p> <p>08.2: Increase specification of all attained efficiency limits.</p>	<p>C8: Extent to which all efficiency constraints are documented.</p> <p>C8.1: Number of efficiency requirements that are documented.</p> <p>C8.2: Proportion of efficiency requirements for which attained limits are specified.</p>	Completeness
<p>09: Promote checking that software products meet required efficiency limits.</p> <p>09.1: Increase reuse of software units with known efficiency limits.</p> <p>09.2: Increase evaluation of efficiency limits on critical parts of software products.</p>	<p>C9: Extent of checking that software products meet required efficiency limits.</p> <p>C9.1: Proportion of software units that are from a component library and have known efficiency limits.</p> <p>C9.2: Proportion of software units affected by an efficiency constraint whose efficiency limits have been experimentally evaluated.</p>	Confirmability

Figure 4.4: continued

It is critically necessary, when using this enumeration approach, to do the seemingly redundant step of converting the objectives into unbiased characteristics. It is the nature of an objective to include some bias, for example, "Minimize complexity of ..." or "Localize the scope of ...". Whereas, a characteristic should be an unbiased statement such as "Degree of complexity of ..." or "Extent of scope of ...". This lack of bias is necessary since, in some situations, it may be desirable to compromise a particular objective so that another can be optimized.

It is also critically necessary to determine, as illustrated in Figure 4.4, subobjectives down to a level leading to measurable characteristics. While many characteristics are directly measurable, some will not be. For example, several relatively well-established metrics, such as Halstead's Software Science metrics or McCabe's Complexity metric, can be used to evaluate the characteristic "degree of complexity of software structures". However, the characteristic "extent to which software structure supports achieving efficiency constraints" cannot be easily measured directly and the objective from which it stems must be successively decomposed until subobjectives are reached that lead to measurable characteristics.

A knowledge of metrics is, therefore, crucial to identifying characteristics. The metrics work done by others, such as at RADC (6) and within the STARS Measurement area, will of course be critical in both guiding the enumeration of characteristics and suggesting characteristics which are not uncovered by enumeration approaches such as discussed here.

The enumeration approach is, in essence, table-driven and can be used to identify and organize an initial set of characteristics in other categories. The matrices for these other categories, discussed in the next subsection, can be used to drive the enumeration approach in order to identify characteristics pertinent to these other categories.

4.4 Defining the Scope of the Other Categories

The organizing matrix shown in Figure 4.3 reflects the goals and principles of software engineering. It relates to the technical aspects of a methodology and, therefore, is most useful in identifying and organizing the technical category of characteristics.

Similar matrices for other categories can be obtained by replacing the goals and principles with those of the other categories of concern. Organizing matrices have been defined in this manner for the management, usage and Ada-compatibility categories and these are discussed in this subsection.

The management category addresses the planning, organizing and controlling of software projects. This area of concern involves consideration of both the management of software products and the people involved in the software creation and evolution process. Consequently, the matrix reflects both purely managerial issues, such as staff scheduling, and more technically influenced issues, such as measuring progress against pre-defined milestones. This matrix is shown in Figure 4.5.

The usage category covers a fairly general area of concern. It includes issues of how a methodology meets the special needs of a particular organization and activities such as the initial acquisition of a methodology by an organization and its subsequent continued use. To achieve this focus, the preliminary matrix given in Figure 4.6 defines goals that relate to the ability to acquire, use and adapt the methodology for a specific project or across a group of projects. The principles defined here indicate some of the considerations that can support meeting these goals.

The Ada-compatibility category concerns how a methodology supports the use of Ada. To some extent, this issue is already dealt with at a conceptual level by the technical matrix. Because Ada was designed to support achieving the goals of software engineering, and these are the goals which are compatibility in the technical matrix, the technical organizing matrix of Figure 4.3 reflects both the technical and Ada-compatibility characteristics of a methodology. However, it is still necessary to reflect how a methodology supports use of Ada at a detailed level and so an additional matrix has been developed to meet this need. This Ada-compatibility matrix is shown in Figure 4.7.

This final matrix reflects how a methodology supports effective use of specific Ada language features. This cannot be done in isolation of considering how these language features are being used. For example, the use of representation specifications for developing software to control peripheral devices will be different from their use in developing pattern recognition software. Therefore, the Ada-compatibility matrix uses major software functions for its goals and the chief Ada language constructs for its principles. Software functions, such as peripheral control and process control, are used in preference to application areas, such as ballistic missile systems and C3 systems since they tend to be better-defined and more specific, whereas application areas typically involve a variety of different software functions. Thus, this matrix supports identifying and organizing characteristics which describe how a methodology supports use of each language construct in developing different types of software functions.

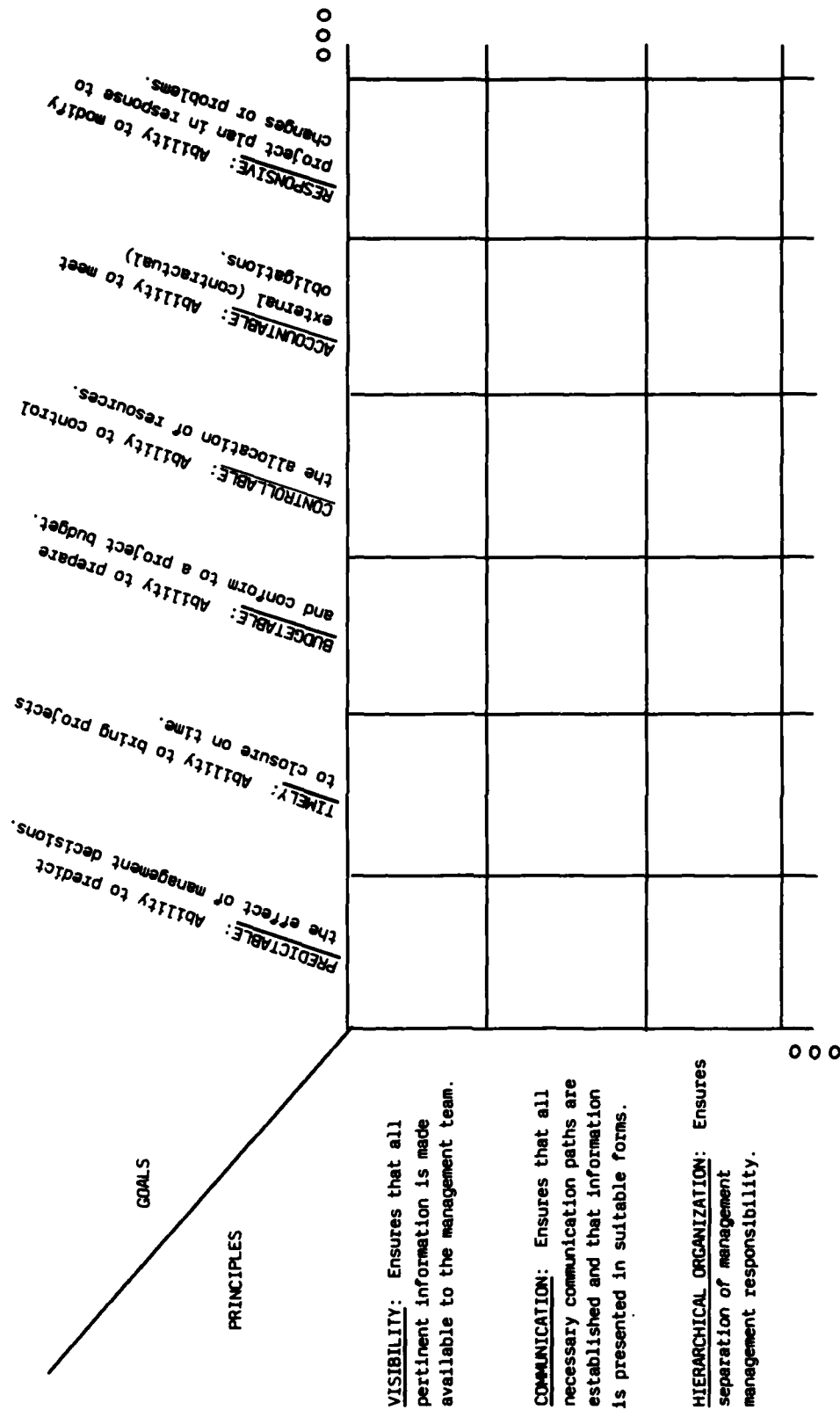


Figure 4.5: Preliminary Matrix for Collections of Management-Related Characteristics

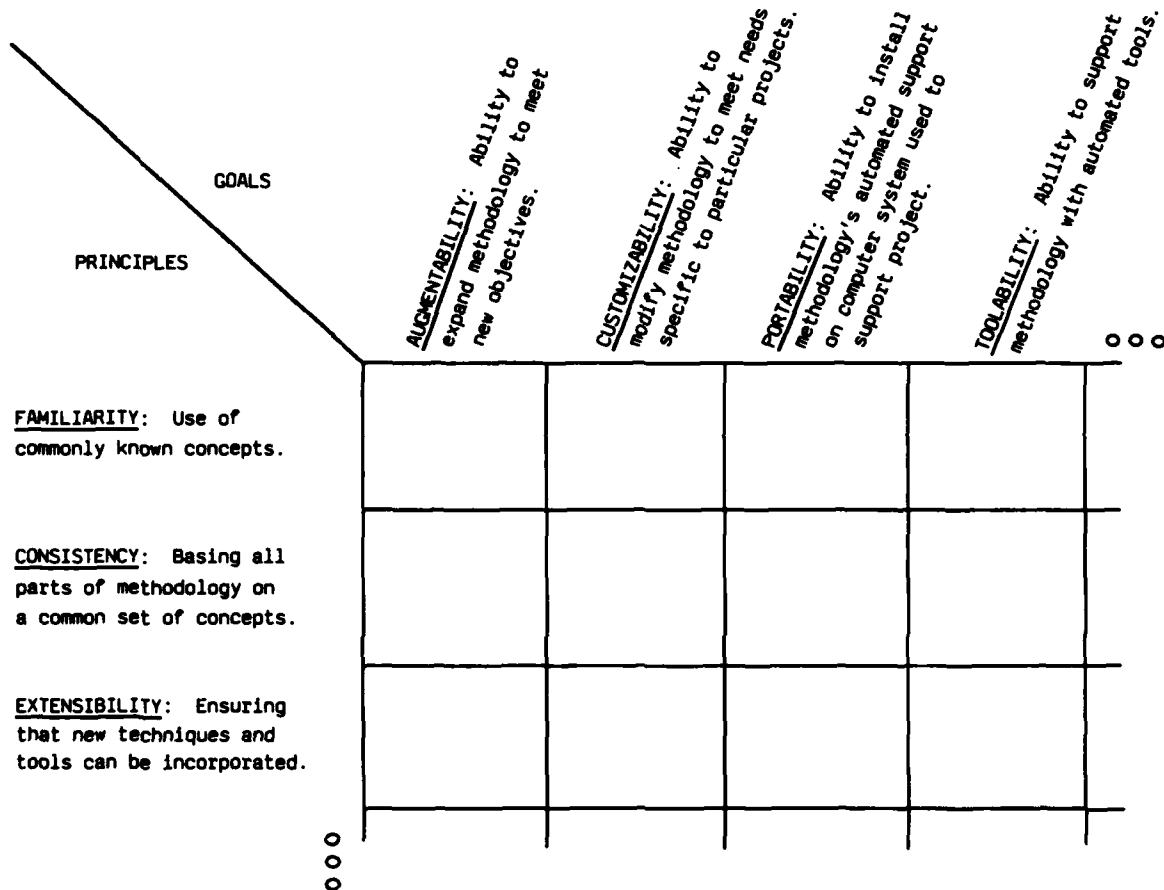


Figure 4.6: Preliminary Matrix for Collections of Usage-Related Characteristics

GOALS	PRINCIPLES													
		PERIPHERAL CONTROL: Support use of conventional and specialized peripheral devices.	SECURITY: Protect from unauthorized access, use, or change.	RECOVERY: Provides failure detection and restart facilities.	PROCESS CONTROL: Receives data and issues commands to control actions.	DATABASE MANAGEMENT: Manages the storage and access of groups of data.	SIMULATION: Used to simulate an environment, scenarios, hardware, etc. to enable evaluation of computer software or hardware.	DATA ACQUISITION: Receives information in real-time and stores for later processing.	DATA PRESENTATION: Formats and transforms data for presentation to humans.	DECISION MAKING: Uses artificial intelligence to evaluate data and provide additional information.	PATTERN AND IMAGE PROCESSING: Used for computer image processing to analyze such information as terrain data.	MESSAGE PROCESSING: Handles input and output of messages and processes message data.	DISTRIBUTED SYSTEMS: Tasks spread among more than one system.	TRANSACTION PROCESSING: Software driven by arrival of data and passage of time.
		PACKAGING: Promotes encapsulating objects and operations and hiding internal details.												
		TYPING: Extends the set of predefined data types and enforces checking.												
		GENERIC: Promotes the development and use of reusable software units.												
		TASKING: Provides a capability for specifying cooperating sequential processes.												
		EXCEPTION HANDLING: Encourages identification and resolution of software failures.												
		REPRESENTATIONS: Provides interfaces to features outside the scope of the language and promotes efficient representations.												

Figure 4.7: Preliminary Matrix for Collections of Ada Compatibility Characteristics

4.5 Current Status of the Characteristics Framework

The technical matrix and enumeration approach have already been used to identify some initial technical characteristics. This initial set of characteristics is incomplete. In particular, the decomposition of objectives to subobjectives is not exhaustive and further decomposition will result in the identification of additional characteristics. In the near future, the characteristics framework will be extended to include additional collections of characteristics in the technical category and to cover the other major categories. This framework will continue to evolve over a span of several years as understanding of the software development and evolution process grows and new types of software methodologies are developed.

5.0 SUMMARY

This document provides a basis for discussing the concerns and issues of software methodology, identifies the pieces of technology needed to be able to classify, evaluate and select among software methodologies, and introduces a flexible framework for organizing the myriad detailed characteristics needed to support these technologies. It provides few solutions but, instead, defines the dimensions of the problems surrounding software creation and evolution and the general nature of the solutions. As such, it lays a groundwork for rationally obtaining and using the technology critically necessary for the disciplined creation and evolution of software.

REFERENCES

- (1) Freeman, P., Wasserman, A.I., "Ada Methodologies: Concepts and Requirements", Ada Joint Program Office, Department of Defense, November 1982.
- (2) "IEEE Standard Glossary of Software Engineering Terminology", IEEE Computer Society, std. 729-1983.
- (3) Department of Defense, "DoD-STD-2167 Defense System Software Development," (Draft), January 30, 1984.
- (4) Boehm, B.W., Gray, T.E., and Seewaldt, T., "Prototyping versus specifying: A multiproject experiment," IEEE Trans. on Software Engineering, SE-10, 3, pp. 290-302, May 1984.
- (5) STARS Measurement Data Item Descriptions (DID's), STARS Measurement Thrust Area, 1984.
- (6) Ross, D.T., Goodenough, J.B., and Irvine, C.A., "Software Engineering: Process, Principles, and Goals," Computer, May 1975.
- (7) "Software Quality Measurement for Distributed Systems," RADC-TR-83-175, July 1983.

APPENDIX A

GLOSSARY

Adaptive Maintenance:	maintenance performed to make a software version usable under a changed set of requirements
Analysis:	the activity of determining whether or not a software version is suitable
Automated Software Environment:	a collection of tools that assists the process of creating and evolving software
Automated System:	a system composed of hardware, software and human components
Characteristic:	see Methodology Characteristic
Characteristic Framework:	a structure for categorizing characteristics
Classification:	determining the characteristics of a methodology and what these characteristics imply about its general type
Coding:	the phase in the life cycle during which data or a software version is represented in a symbolic form that can be accepted by a processor
Conception:	the point in time at which there is an initial perception of need for a software version
Corrective Maintenance:	maintenance performed to overcome identified faults
Coverage:	extent to which a methodology covers the full life cycle of some type of version
Criteria:	use-oriented, high-level attributes useful in deciding acceptability
Data Structuring Methodology:	methodology that focuses initial attention on the definition of system inputs and outputs

Definition: the point in time at which a software version is described by a document defining the problem to be solved and the general nature of and requirements upon its solution

Delivery: the point in a life cycle at which a software version is released for integration into the automated system of which it is a part

Deployment: The point in a life cycle at which a software version is released for operational use

Detailed Design: the phase in a life cycle during which the preliminary design is refined and expanded to contain more detailed descriptions of the processing logic, data structures, and data definitions, to the extent that the design is sufficiently complete to be implemented

Development: the process by which user needs are transformed into a software version that can be delivered

Evaluation: determining whether a methodology meets certain criteria

Freezing: the point in a life cycle at which it is decided that no further changes will be made to an operational software version

Implementation: see coding

Installation: the process by which a software version is integrated into its operational environment and tested in this environment to ensure that it performs as required

Life Cycle: the period of time from the initial perception of need for a software version to its retirement

Maintenance:	modification of a software version after delivery to correct faults, improve performance or other attributes, or meet new requirements
Measure:	a quantity that can be evaluated to determine whether or not a methodology meets a particular criteria
Method:	a set of rules, guidelines, and techniques for carrying out a process
Methodology:	a general philosophy for carrying out a process; comprised of procedures, principles, and practices
Methodology Characteristic:	a detailed attribute that can be used to describe a methodology
Metric:	a quantity that can be evaluated to determine whether or not a methodology has a particular characteristic
Model:	a representation which specifies some but not all of an entity's attributes
Object:	an encapsulation of data and/or processing activity which reflects some entity in the software or its operational environment
Object-Oriented Methodology:	a methodology that represents the organization of a piece of software as a layering of successively more detailed objects
Operation:	use of a version in its operational environment
Operation and Maintenance:	use of a software system in its operational environment; involves monitoring for satisfactory performance and modification as necessary to correct problems or respond to changed requirements

**Perfective
Maintenance:**

maintenance performed to improve performance, maintainability, or other software attributes

Phase:

a period of time during a life cycle

**Product-Oriented
Methodology:**

a methodology that is defined primarily by specifying the intermediate and final products to be produced; definition of the products is usually accompanied by the definition of phases where each phase is focused on the preparation of one or more of the products

**Post-Deployment
Support:**

see Maintenance

Preliminary Design:

the phase in a life cycle during which alternatives are analyzed and the general architecture of a software version is defined; typically includes definition and structuring of modules and data, definition of interfaces, and preparation of timing and sizing estimates

Product:

results created by a process

Prototype:

an instance of a software version that does not exhibit all the properties of the final system; usually lacking in terms of functional or performance attributes

**Prototyping
Methodology:**

a methodology that organizes the creation and evolution of a software version as a series of prototypes

Release:

a software version that is delivered for integration into an automated system

**Requirements
Definition:**

the phase in the life cycle during which the requirements, such as the functional and performance capabilities, are defined

Retirement:	the point in a life cycle at which a software version is removed from service
Scope:	extent to which a methodology disciplines the creation and evolution of a software system rather than just the individual releases or variants
Selection:	picking a methodology, or set of alternative methodologies, for use on a specific project
Software:	the executable code, all of its associated documentation and documents that trace the history of its creation and evolution
Software System:	a component of an automated system that is realized as executable code
Specification:	the point in time at which a version is described in a document that defines, in a relatively complete, precise, and verifiable manner, the requirements of a software version
Technology:	collection of techniques and knowledge underlying some process
Test and Integration:	the phase in a life cycle during which the conformance of the version to its requirements is assessed and the version is integrated into the larger (software or automated) system of which it is a part
Tool:	software which assists in carrying out a task or activity
Variant:	any of those versions of a software system which are prepared in the course of developing a release
Validation:	analyzing a version to assure that it meets user needs
Verification:	analyzing a version to assure that it meets its requirements
Version:	any instance of a software system

DISTRIBUTION LIST

STARS OFFICE

Dr. Ed Lieblein (3)
STARS Joint Program Office
3D139 (1211 Fern, C107)
Pentagon
Washington, D.C. 20301-3081

MCT CHAIRS

Mr. Lou Chmura
Code 7592
Naval Research Lab
4555 Overlook Avenue, S.W.
Washington, D.C. 20375-5000

Mr. Peter Fonash
AMC/BAM (AMCDE-SB)
Alexandria, Virginia 22337

Mr. Larry Lindley
Code D/072.2
Naval Avionics Center
6000 E 21st Street
Indianapolis, Indiana 46218

Mr. Kenneth Rowe
6200 Harris Heights Ave.
Glen Burnie, MD 21061

Mr. George Sumrall
18 Manor Drive
Neptune, NJ 07753

CSED REVIEW PANEL

Dr. Dan Alpert, Director
Center for Advanced Study
University of Illinois
912 W. Illinois Street
Urbana, IL 61801

Dr. Barry W. Boehm
TRW Defense Systems Group
MS 02-2304
One Space Park
Redondo Beach, CA 90278

Dr. Ruth Davis
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, VA 22201

Dr. Larry E. Druffel
Rational Machines
1501 Salado Drive
Mountain View, CA 94043

Mr. Neil S. Eastman, Manager
Software Engineering & Technology
IBM Federal Systems Division
6600 Rockledge Drive
Bethesda, MD 20817

Admiral Noel Gayler, USN, Retired
1250 S. Washington Street
Alexandria, VA 22314

Dr. Charles E. Hutchinson
Dean, Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

Mr. Oliver Selfridge
45 Percy Road
Lexington, MA 02173

Dr. Harrison Shull, Chancellor
University of Colorado
Campus Box B-17
301 Regent Administration Center
Boulder, CO 80309

Dr. Robert L. Sproull
President Emeritus
University of Rochester
Rochester, NY 14627

DoD-IDA Management Office
1801 N. Beauregard St.
Alexandria, VA 22311

IDA

Mr. Seymour Deitchman, HQ
Mr. Robin Pirie, HQ
Dr. John F. Kramer, Director, CSED
Mr. Gil Berglass, CSED
Ms. Anne Douville, CSED
Ms. Audrey Hook, CSED
Mr. Joseph Hrycyszyn, CSED
Mr. Robert Knapper, CSED
Ms. Catherine W. McDonald, CSED (10)
Mr. Richard Morton, CSED
Ms. Sarah Nash, CSED
Ms. Katydean Price, CSED (2)
Mr. Samuel T. Redwine, Jr., CSED (5)
Mr. Clyde Roby, CSED
Dr. John Salasin, CSED
Ms. Francoise Youssefi, CSED
IDA Control & Distribution Vault (10)

END

FILMED

2-86

DTIC